Criando um Deskbar: Aplique POO e refatoração na prática

Clube Delphi

Ano 9 :: Edição 134 :: R$ 11,90

DevMedia

ISSN 1517990-7

Efeitos Gráficos
Melhorando a aparência de aplicações Delphi

Boas Práticas
POO - Dominando o uso de Interfaces

Padrões de Projeto
Façade e Strategy

UI SEPARADA

Conheça o MVC, MVP e MVVM

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Sumário

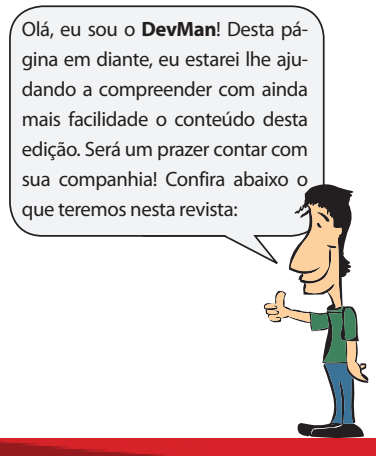
Boas Práticas

Boas Práticas — **06 – Padrões de Projeto - Façade e Strategy**
[*Renato Jose Groffe*]

Boas Práticas — **14 – Criando uma deskbar**
[*Vitor Luiz Rubio*]

Boas Práticas — **30 – Os padrões de arquitetura MVC, MVP e MVVM**
[*Fabrício Hissao Kawata*]

Boas Práticas — **39 – POO - Dominando o uso de Interfaces**
[*Vitor Luiz Rubio*]



Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 12 - 134ª Edição 2011 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Leandro Daniel
contato@leandrodaniel.com

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Jornalista Responsável
Kaline Dolabella - JP24185

Atendimento ao Leitor
A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/mancad
(21) 3382-5038

Publicidade
Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição
FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Existe uma boa prática que é comum a grande maioria de tipo de software: separar a interface de usuário da lógica de negócio. Os usuários de um sistema interagem com o mesmo pela camada de apresentação (Presentation Layer, UI), nela estão os controles e componentes visuais necessários para apresentação e solicitação de informações. Essa camada por sua vez se comunica com uma camada de serviço, que simplifica para a UI as rotinas através da criação de um API que abstraia um processo maior que envolva outras ações. Assim, a interface acessando apenas os serviços, a implementação desses serviços pode mudar sem necessariamente exigir mudança na UI. É muito importante que o código dessas camadas não se misture. Como a lógica da interface não está codificada e amarrada nos controles visuais utilizados, pode-se facilmente trocar esses controles sem influenciar o funcionamento de uma tela ou página por exemplo.

Outro ponto favorável é a possibilidade de se testar a aplicação de forma automatizada. Ao fazer a separação é possível testar o comportamento da UI sem ter a necessidade de executar o software propriamente dito. Mas, existe um porém. Ao separar de forma correta não só a camada de interface, mas todas as camadas lógicas necessárias, há um aumento na complexidade do código, o que pode ser um problema para o entendimento do todo por parte de desenvolvedores menos experientes. Para alcançar essa separação vários estudos ao longo dos anos foram feitos. Esses estudos e implementações levaram à criação dos chamados Padrões de Projeto. Um padrão de projeto é uma solução para um problema recorrente.

A separação da camada de interface é um problema recorrente em todo software. Assim, foram desenvolvidos alguns padrões que cuidam disso, os mais conhecidos são: Model-View-Controller (MVC) e Model-View-Presenter (MVP). Esses são padrões bem genéricos, que podem ser implementados em qualquer tecnologia, já que um padrão não define código e sim, a forma como solucionar o problema.

O WPF surgiu como base para desenvolvimento de aplicativos Windows e que gradativamente vem substituindo a plataforma anterior, o Windows Forms. Com o WPF novos recursos foram criados, de tal forma que eles poderiam ser utilizados para facilitar a separação de código. Assim surgiu o padrão Model-View-ViewModel ou simplesmente MVVM, que nada mais é do que um padrão de separação de interface que faz uso dos recursos do WPF e mais recentemente do Silverlight.

Nesta edição, você conhecerá cada um destes padrões, bem como construir um exemplo prático utilizando o Delphi, na matéria do Fabrício Kawata.

Boa leitura!

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Linux
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET,C#)
guintherpauli.blogspot.com - twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 0,99

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Utilizando os padrões de

Resumo DevMan

De que trata o artigo:

Este artigo procura discutir o uso de padrões de projeto (do inglês, Design Patterns) em aplicações de software, apresentando para isto como os padrões conhecidos como Strategy e Façade podem ser empregados na construção de sistemas flexíveis e bem estruturados.

Em que situação o tema é útil:

Design patterns são técnicas concebidas para a solução de problemas inerentes à construção de softwares. O pattern conhecido como Strategy é uma alternativa que pode ser empregada, via de regra, em situações que envolvam comportamentos variáveis para um determinado aspecto. Já o padrão Façade tem por finalidade simplificar operações complexas que envolvam a utilização de diferentes objetos ao mesmo tempo, de forma que classes consumidoras que dependam disto estejam isentas de conhecer a fundo todos os detalhes necessários.

Utilizando os padrões de projeto Strategy e Façade:

No geral, os padrões Strategy e Façade são aplicados, respectivamente, no tratamento de comportamentos variáveis e na simplificação de construções que manipulem diversos tipos de objetos e operações.

Muitos desenvolvedores acabam se deparando, ao longo de uma série de projetos, com determinadas situações que se repetem com frequência. Em muitos destes casos é possível que outros profissionais já tenham passado por circunstâncias semelhantes e, a partir disso, disponibilizado soluções que buscam resolver estas questões.

Tais soluções são conhecidas como design patterns (padrões de projetos), sendo normalmente voltadas à solução de problemas específicos dentro do desenvolvimento de software. Tomando-se por base um aspecto que varia dentro de uma aplicação, o pattern Strategy é uma técnica que fornece meios para o controle dos diversos comportamentos possíveis neste caso. O padrão Façade, por outro lado, encapsula a complexidade advinda da interação entre uma série de objetos, facilitando assim o acesso de classes consumidoras a tais operações.

Design patterns (termo conhecido também em português como "padrões de projetos") são técnicas concebidas com a finalidade de resolver problemas específicos que podem, por sua vez, aparecer com certa frequência em atividades relacionadas ao desenvolvimento de sistemas. A adoção de padrões procura fornecer, via de regra, meios para o reuso de soluções de software cuja validade já foi comprovada anteriormente.

Mudanças serão sempre uma constante dentro do desenvolvimento de software. Alterações em atendimento a uma nova demanda de negócio ou mesmo a evolução de funcionalidades pré-existentes são aspectos com os quais os envolvidos em projetos de sistemas sempre irão se deparar. Além disso, novas soluções de software são criadas a todo momento, sendo isto uma causa direta do cenário de mudanças constantes em que estão envoltas as organizações.

Design patterns podem, em ambas as situações relatadas, fornecer meios para uma melhor estruturação de aplicações,

projeto Strategy e Façade

Renato José Groffe

flexibilizando o desenvolvimento e auxiliando de maneira significativa para tornar mais simples futuras atividades de manutenção.

Muitos dos padrões de projeto empregados no desenvolvimento de software fazem uso de conceitos de orientação a objetos, sendo que o correto uso dos mesmos contribui para a assimilação de boas práticas OO na construção de sistemas.

É importante frisar que design patterns não são técnicas restritas a uma determinada plataforma ou linguagem de programação. Por serem soluções gerais em resposta a necessidades de projetos, bem como representarem experiências comprovadas e bem sucedidas, muitos padrões podem estar sendo empregados normalmente em linguagens como Delphi, C#, VB.NET, Java etc.

Levando em conta a ampla diversidade de tipos de aplicações de software existentes atualmente, não é de se admirar que tenham sido desenvolvidos patterns para os mais diferentes tipos de situações. Os esforços para a determinação de padrões iniciaram-se na área de software, ainda na década de 90, com a publicação de 23 padrões conhecidos como GoF.

A finalidade deste artigo é apresentar algumas das vantagens do uso de patterns na construção de aplicações. Além disso, serão abordados 2 padrões GoF: Strategy e Façade. O pattern Strategy permite que se separem comportamentos variáveis em classes que encapsulam os algoritmos correspondentes, dispensando com isto a necessidade dos consumidores de uma funcionalidade conhecerem a fundo todas os comportamentos possíveis para a mesma. Já o padrão Façade possibilita que um conjunto de ações envolvendo diversos objetos possa ser encapsulado, de forma que os utilizadores da funcionalidade em questão não precisem realizar uma série de ações complexas para produzir o efeito esperado.

Objetivos e benefícios do uso de padrões

A aplicação de patterns em um projeto de software tem por principais objetivos:

- Evitar que se obtenham extensas construções de código durante a construção de uma solução;
- Possibilitar que o código obtido seja de fácil compreensão por parte de desenvolvedores que venham a trabalhar com o mesmo;
- Uma maior flexibilidade para se alterar uma aplicação quando da necessidade de se efetuarem mudanças em atendimento a requisitos de negócio;
- Divisão equilibrada de responsabilidades entre as diferentes classes que compõem um projeto;
- O reuso de componentes de software, não ficando isto restrito a uma única aplicação, mas com o reaproveitamento de funcionalidades em diversas soluções.

Nota do DevMan

GoF é a sigla em inglês para "Gang of Four", sendo um apelido dado ao trabalho conjunto de 4 autores (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides), os quais documentaram 23 padrões no famoso livro "Design Patterns: Elements of Reusable Object-Oriented Software". Tais patterns foram especificados na época tomando-se como base exemplos desenvolvidos em C++ e Smalltalk; no entanto, a ampla abrangência destes padrões possibilitou que os mesmos sejam utilizados ainda hoje, auxiliando desenvolvedores Delphi, .NET e Java na resolução de uma série de problemas recorrentes no cotidiano.

Podem ser listados como benefícios do uso de padrões de projetos:

- Uma maior padronização em atividades relacionadas à codificação de funcionalidades;
- A possibilidade de reuso de uma mesma arquitetura em variados projetos;
- Uma maior coesão nos diferentes componentes de uma solução, com cada classe acumulando apenas funções que condizem com o propósito para o qual a mesma foi criada inicialmente;
- Maior facilidade na manutenção de uma aplicação, com menores impactos quando surgir a necessidade de se modificar algo, considerando para isto um baixo acoplamento entre os diversos componentes que constituam a solução;
- A redução de custos em novos projetos, a partir do reaproveitamento de componentes e funcionalidades já construídos anteriormente;
- A obtenção de soluções concebidas segundo boas práticas de OO, haja visto a ênfase dada pelos diferentes design patterns à utilização de técnicas já consolidadas dentro da orientação a objetos;
- Por contarem com ampla documentação, padrões de projeto podem ser aprendidos com facilidade, fato que melhora a interação e o compartilhamento de conhecimentos entre desenvolvedores dentro de uma mesma equipe.

Tipos de patterns GoF

Levando em conta o tipo de problema que procuram solucionar, os patterns GoF podem ser classificados em:

- Padrões de criação: envolvem técnicas para a instanciação de objetos;

• Padrões estruturais: estabelecem maneiras para se trabalhar com a composição de objetos;
• Padrões comportamentais: estabelecem regras para a interação entre classes ou, ainda, como ficarão as responsabilidades das mesmas.

Na **Tabela 1** encontram-se listados os padrões GoF existentes, com os mesmos estando agrupados por tipo.

Tipo	Patterns
Padrões de criação	Abstract factory, Builder, Factory method, Prototype e Singleton.
Padrões estruturais	Adapter, Bridge, Composite, Decorator, Façade, Flyweight e Proxy.
Padrões comportamentais	Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method e Visitor.

Tabela 1. Patterns GoF classificados por tipo

Padrões Strategy e Façade: uma visão geral

Strategy (**Figura 1**) é um pattern comportamental, sendo também conhecido pelo nome de Policy. Neste padrão uma família de algoritmos (classes concretas) representando um tipo de comportamento é definida. Cada um destes algoritmos é implementado de uma maneira específica, representando com isto uma variação possível para o comportamento que se está considerando. Classes que dependam de tal tipo de comportamento ficam isentas de conhecer as diferentes implementações para o mesmo, sendo isto consequência da utilização de um supertipo (interface ou classe abstrata) que define todos os aspectos possíveis.

O uso do pattern Strategy pode ser de grande valia em cenários que envolvam variações para um mesmo aspecto. Um exemplo clássico disto são aplicações que realizem o cálculo de impostos: um mesmo tipo de imposto pode ser calculado de "n" maneiras, com isto acontecendo em virtude das diversas regulamentações que podem existir.

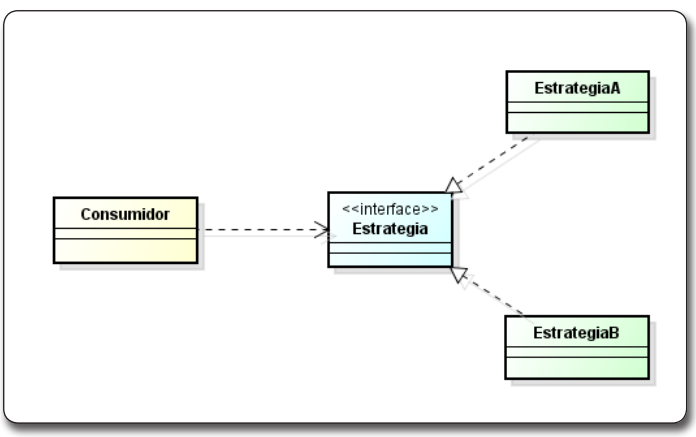


Figura 1. Representação esquemática do pattern Strategy

Já Façade (**Figura 2**) é um padrão estrutural que tem por objetivo simplificar um conjunto de interações envolvendo diversas classes, disponibilizando para isso um tipo dotado de operações que simplificam a execução de tais ações. Logo, pode-se concluir que a função deste pattern está no fato de que o mesmo possibilita, basicamente, desconectar uma classe consumidora dos tipos dos quais esta última depende.

A utilização do pattern Façade contribui para diminuir o acoplamento de uma classe aos vários tipos que são necessários para a utilização da mesma. O baixo acoplamento é uma característica desejável sobretudo em softwares dotados de um alto grau de complexidade, uma vez que possibilita que novas alterações sejam introduzidas em um ponto da aplicação sem que as demais partes desta sofram um grande impacto.

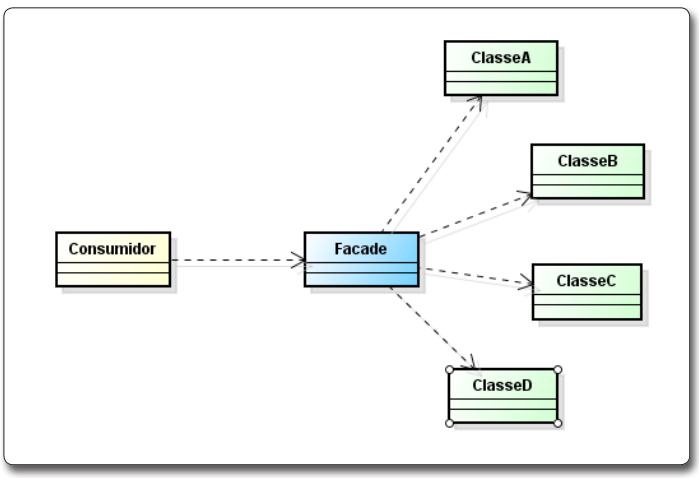


Figura 2. Representação esquemática do pattern Façade

Exemplo de utilização dos patterns Strategy e Façade

O desenvolvimento da aplicação demonstrada neste artigo foi realizado através da utilização do Embarcadero Delphi XE, sendo que através da mesma será discutido o uso dos patterns Strategy e Façade.

A fim de se cumprir com tal objetivo será criado um projeto do tipo "VCL Forms Application" chamado TesteStrategyFacade, sendo que o mesmo envolve, de maneira simplificada, a execução de cálculos de impostos para produtos de uma nota fiscal. Considerando o cenário de complexidade que envolve a área de tributos, o uso dos dois padrões de projetos aqui abordados oferece um caminho para a obtenção de soluções de software mais flexíveis.

O cálculo de impostos em notas fiscais envolve numerosas regras, sendo que para efeitos de simplificação estão sendo considerados os seguintes pontos:
• Para se calcular o IPI será aplicada a alíquota deste tributo sobre o total de um produto;
• O valor do IPI também fará parte da base de cálculo do ICMS;
• A base de cálculo do ICMS poderá ser reduzida a um percentual informado pelo usuário, sendo obtida a partir do somatório do valor de um item com o IPI que incidirá sobre o mesmo;
• O valor do ICMS é obtido aplicando-se a alíquota deste imposto sobre a base, que pode ou não ter sido reduzida.

Criação das classes para cálculo de impostos

Na **Listagem 1** são apresentadas as classes TInformacoesIPIItem, TInformacoesICMSItem e TImpostosItem. Os tipos TInformacoesIPIItem e TInformacoesICMSItem são utilizados para fornecer informações para o cálculo do IPI e do ICMS, respectivamente. Já a classe TImpostosItem, por sua vez, será usada para retornar as bases de cálculo e os impostos correspondentes.

A classe TCalculoIPI (**Listagem 2**) é responsável por calcular a base o valor do IPI, fazendo uso para isto do tipo TInformacoesIPIItem já definido anteriormente.

A interface ICalculoBaseICMS (**Listagem 3**) servirá de base para a demonstração do pattern Strategy. A mesma define um conjunto de operações que deverão ser implementadas pelas classes responsáveis pelo cálculo da base de ICMS (com ou sem redução). Pelo fato da forma como se obtém a base deste imposto ser variável, o uso de Strategy fornece uma solução prática para um comportamento que pode mudar de acordo com uma série de condições.

Listagem 1. Classes TInformacoesIPIItem, TInformacoesICMSItem e TImpostosItem

```
unit InformacoesImpostos;

interface

type

 TInformacoesIPIItem = class
 private
  rValorItem: Real;
  rValorAliquotaIPI: Real;
 public
  property ValorItem: Real read rValorItem write rValorItem;
  property ValorAliquotaIPI: Real read rValorAliquotaIPI write rValorAliquotaIPI;
 end;

 TInformacoesICMSItem = class
 private
  rValorItem: Real;
  rValorAliquotaICMS: Real;
  rValorPercReducaoAliqICMS: Real;
  rValorIPI: Real;
 public
  property ValorItem: Real read rValorItem write rValorItem;
  property ValorAliquotaICMS: Real read rValorAliquotaICMS
   writerValorAliquotaICMS;
  property ValorPercReducaoAliqICMS: Real read rValorPercReducaoAliqICMS
  write rValorPercReducaoAliqICMS;
  property ValorIPI: Real read rValorIPI write rValorIPI;
 end;

 TImpostosItem = class
 private
  rValorBaseICMS: Real;
  rValorICMS: Real;
  rValorBaseIPI: Real;
  rValorIPI: Real;
 public
  property ValorBaseIPI: Real read rValorBaseIPI write rValorBaseIPI;
  property ValorIPI: Real read rValorIPI write rValorIPI;
  property ValorBaseICMS: Real read rValorBaseICMS write rValorBaseICMS;
  property ValorICMS: Real read rValorICMS write rValorICMS;
 end;

implementation

end.
```

Listagem 2. Classe TCalculoIPI

```
unit CalculoIPI;

interface

uses InformacoesImpostos;

type

 TCalculoIPI = class
 private
  vInformacoesIPIItem: TInformacoesIPIItem;
 public
  procedure SetInformacoesIPIItem(informacoesIPIItem: TInformacoesIPIItem);
  function GetValorBaseIPI: Real;
  function GetValorIPI: Real;
 end;

implementation

{ TCalculoIPI }

procedure TCalculoIPI.SetInformacoesIPIItem(
 informacoesIPIItem: TInformacoesIPIItem);
begin
 vInformacoesIPIItem:= informacoesIPIItem;
end;

function TCalculoIPI.GetValorBaseIPI: Real;
begin
 if (vInformacoesIPIItem.ValorAliquotaIPI > 0) then
  Result:= vInformacoesIPIItem.ValorItem
 else
  Result:= 0;
end;

function TCalculoIPI.GetValorIPI: Real;
begin
 if (vInformacoesIPIItem.ValorAliquotaIPI > 0) then
 begin
  Result:= (GetValorBaseIPI *
        vInformacoesIPIItem.ValorAliquotaIPI / 100);
 end
 else
  Result:= 0;
end;

end.
```

Listagem 3. Interface ICalculoBaseICMS

```
unit CalculoBaseICMS;

interface

uses InformacoesImpostos;

type
 ICalculoBaseICMS = interface
  procedure SetInformacoesICMSItem(informacoesICMSItem: TInformacoes
  ICMSItem);
  function GetValorBaseICMS: Real;
 end;

implementation

end.
```

A classe TCalculoBaseICMSNormal (**Listagem 4**) tem por função calcular a base do ICMS sem a aplicação do percentual de redução da mesma. A base reduzida será obtida, por sua vez, através do uso da classe TCalculoBaseICMSReduzida (**Listagem 5**). Estes tipos fazem uso da classe TInformacoesICMSItem, recebendo uma instância desta última por meio da procedure SetInformacoesICMSItem. TCalculoBaseICMSNormal e TCalculoBaseICMSReduzida implementam a interface ICalculoBaseICMS, o que é premissa na aplicação de exemplo para o uso do pattern Strategy.

Importante ressaltar, ainda, que os dois tipos de cálculo de base de ICMS estão herdando a classe TInterfacedObject. Toda interface em Delphi deriva de IUnknown: a implementação de uma interface acaba obrigando que alguns métodos que existem em IUnknown também sejam definidos. A utilização de TInterfacedObject dispensa uma classe da necessidade de implementar métodos de IUnknown.

É interessante notar que a maioria dos patterns visa a eliminação ou controle da duplicação de código. A duplicação de código é um dos maiores inimigos do desenvolvedor, pois manter um programa com esse tipo de problema é uma atividade trabalhosa e muito propensa a erros. A complexidade de uma aplicação cresce de forma descontrolada toda vez que não reutilizamos um determinado código ou deixamos de aplicar boas práticas de programação.

Por outro lado, vale ressaltar que o uso indiscriminado de patterns gera complexidade desnecessária ao software, o que pode ser tão desastroso quanto a duplicação de código. A melhor saída é procurar estudar bastante os padrões de projeto e saber, sempre que possível, qual a opinião de desenvolvedores mais experientes.

Na **Listagem 6** encontra-se a definição do tipo TCalculoICMS. Esta classe tem por objetivo calcular a base e o valor de ICMS de um item, sendo que para isto está sendo feito uso de uma instância do tipo TInformacoesICMSItem (a qual contém informações que servem de base para se calcular o ICMS), bem como de uma referência para um objeto que implemente a interface ICalculoBaseICMS (TCalculoICMS não possui conhecimento direto das implementações de ICalculoBaseICMS, já que estamos implementando o padrão Strategy justamente para evitar isto).

Listagem 4. Classe TCalculoBaseICMSNorma

```
|
unit CalculoBaseICMSNormal;
interface
uses CalculoBaseICMS, InformacoesImpostos;
type
 TCalculoBaseICMSNormal = class(TInterfacedObject, ICalculoBaseICMS)
 private
  vInformacoesICMSItem: TInformacoesICMSItem;
 public
  procedure SetInformacoesICMSItem(informacoesICMSItem: TInformacoes
  ICMSItem);
  function GetValorBaseICMS: Real;
 end;
implementation
{ TCalculoICMSNormal }
procedure TCalculoBaseICMSNormal.SetInformacoesICMSItem(
 informacoesICMSItem: TInformacoesICMSItem);
begin
 vInformacoesICMSItem:= informacoesICMSItem;
end;

function TCalculoBaseICMSNormal.GetValorBaseICMS: Real;
begin
 Result:= vInformacoesICMSItem.ValorItem +
       vInformacoesICMSItem.ValorIPI;
end;
end.
```

Listagem 5. Classe TCalculoBaseICMSReduzida

```
unit CalculoBaseICMSReduzida;

interface

uses CalculoBaseICMS, InformacoesImpostos;

type

 TCalculoBaseICMSReduzida = class(TInterfacedObject, ICalculoBaseICMS)
 private
  vInformacoesICMSItem: TInformacoesICMSItem;
 public
  procedure SetInformacoesICMSItem(
   informacoesICMSItem: TInformacoesICMSItem);
  function GetValorBaseICMS: Real;
 end;

implementation

{ TCalculoBaseICMSReduzida }

procedure TCalculoBaseICMSReduzida.SetInformacoesICMSItem(
 informacoesICMSItem: TInformacoesICMSItem);
begin
 vInformacoesICMSItem:= informacoesICMSItem;
end;

function TCalculoBaseICMSReduzida.GetValorBaseICMS: Real;
begin
 Result:= (vInformacoesICMSItem.ValorItem +
       vInformacoesICMSItem.ValorIPI) *
      (100 - vInformacoesICMSItem.ValorPercReducaoAliqICMS) / 100;
end;

end.
```

A classe TImpostosFacade (**Listagem 7**) corresponde à implementação do padrão Façade. Este tipo foi estruturado, basicamente, de uma maneira que as classes consumidoras do mesmo não possuam um conhecimento direto de todos os outros objetos envolvidos no processo. Logo, consumidores da funcionalidade disponibilizada pelo método CalcularImpostosItem de TImpostosFacade não terão conhecimento de como classes as TCalculoIPI, TCalculoICMS, TCalculoBaseICMSReduzida e TCalculoBaseICMSNormal estão sendo empregadas, por exemplo.

A base e o valor do IPI estão sendo calculados por meio de um objeto do tipo TCalculoIPI (que recebe para isto uma instância de TInformacoesIPIItem via método SetInformacoesIPIItem).

Listagem 6. Classe TCalculoICMS

```
unit CalculoICMS;

interface

uses CalculoBaseICMS, InformacoesImpostos;

type

 TCalculoICMS = class
 private
  vInformacoesICMSItem: TInformacoesICMSItem;
  vCalculoBaseICMS: ICalculoBaseICMS;
 public
  procedure SetInformacoesICMSItem(informacoesICMSItem: TInformacoes
   ICMSItem);
  procedure SetCalculoBaseICMS(calculoBaseICMS: ICalculoBaseICMS);
  function GetValorBaseICMS: Real;
  function GetValorICMS: Real;
 end;

implementation

{ TCalculoICMS }

procedure TCalculoICMS.SetCalculoBaseICMS(calculoBaseICMS: ICalculoBaseICMS);
begin
 vCalculoBaseICMS:= calculoBaseICMS;
end;

procedure TCalculoICMS.SetInformacoesICMSItem(
 informacoesICMSItem: TInformacoesICMSItem);
begin
 vInformacoesICMSItem:= informacoesICMSItem;
end;

function TCalculoICMS.GetValorBaseICMS: Real;
begin
 Result:= vCalculoBaseICMS.GetValorBaseICMS;
end;

function TCalculoICMS.GetValorICMS: Real;
begin
 Result:= (GetValorBaseICMS *
       vInformacoesICMSItem.ValorAliquotaICMS / 100);
end;

end.
```

Listagem 7. Classe TImpostosFacade

```
unit ImpostosFacade;
interface
uses InformacoesImpostos;
type
 TImpostosFacade = class
 public
  function CalcularImpostosItem(
   rValorItem,
   rAliquotaICMS,
   rPercReducaoICMS,
   rAliquotaIPI: Real): TImpostosItem;
 end;
implementation
uses CalculoIPI, CalculoBaseICMS, CalculoBaseICMSNormal,
      CalculoBaseICMSReduzida, CalculoICMS;
     { TImpostosFacade }
function TImpostosFacade.CalcularImpostosItem(
 rValorItem,
 rAliquotaICMS,
 rPercReducaoICMS,
 rAliquotaIPI: Real): TImpostosItem;
var
 impostosItem: TImpostosItem;
 informacoesIPI: TInformacoesIPIItem;
 calculoIPI: TCalculoIPI;
 informacoesICMS: TInformacoesICMSItem;
 calculoBaseICMS: ICalculoBaseICMS;
 calculoICMS: TCalculoICMS;
begin
   impostosItem:= TImpostosItem.Create;
   informacoesIPI:= TInformacoesIPIItem.Create;
   informacoesIPI.ValorItem:= rValorItem;
   informacoesIPI.ValorAliquotaIPI:= rAliquotaIPI;
   calculoIPI:= TCalculoIPI.Create;
   calculoIPI.SetInformacoesIPIItem(informacoesIPI);
   impostosItem.ValorBaseIPI:= calculoIPI.GetValorBaseIPI;
   impostosItem.ValorIPI:= calculoIPI.GetValorIPI;
   informacoesICMS:= TInformacoesICMSItem.Create;
   informacoesICMS.ValorItem:= rValorItem;
   informacoesICMS.ValorIPI:= impostosItem.ValorIPI;
   informacoesICMS.ValorAliquotaICMS:= rAliquotaICMS;
   informacoesICMS.ValorPercReducaoAliqICMS:= rPercReducaoICMS;
   if (rPercReducaoICMS > 0) then
      calculoBaseICMS:= TCalculoBaseICMSReduzida.Create
   else
      calculoBaseICMS:= TCalculoBaseICMSNormal.Create;
      calculoBaseICMS.SetInformacoesICMSItem(informacoesICMS);
      calculoICMS:= TCalculoICMS.Create;
      calculoICMS.SetInformacoesICMSItem(informacoesICMS);
      calculoICMS.SetCalculoBaseICMS(calculoBaseICMS);
      impostosItem.ValorBaseICMS:= calculoICMS.GetValorBaseICMS;
      impostosItem.ValorICMS:= calculoICMS.GetValorICMS;
      Result:= impostosItem;
   end;
end.
```

As mesmas informações para o ICMS são obtidas através de um referência para TCalculoICMS (a qual depende de um objeto do tipo TInformacoesICMSItem, fornecido por meio do método SetInformacoesICMSItem).

Além disso, a instância da classe TCalculoICMS está recebendo um objeto declarado como sendo do tipo ICalculoBaseICMS (através da operação SetCalculoBaseICMS): caso o percentual de redução da base de cálculo seja maior do que zero, vincula-se uma instância de TCalculoBaseICMSReduzida à variável calculoBaseICMS; do contrário, opta-se pela utilização de TCalculoBaseICMSNormal. Como se está empregando o padrão Strategy para o cálculo da base de ICMS, a classe TCalculoICMS é dispensada da necessidade de conhecer detalhes específicos referentes à forma de se obter esta informação.

Uma vez que todos os passos para o cálculo do IPI e do ICMS tenham sido executados, o método CalcularImpostosItem devolverá um objeto do tipo TImpostosItem, com o mesmo contendo a base e os valores para estes dois impostos.

Implementação da tela para testes

Uma vez que o processo de codificação das classes para a manipulação dos impostos tenha finalizado, deve-se prosseguir com a implementação da tela que será empregada para os testes das funcionalidades descritas neste artigo. O formulário principal da aplicação deve ser renomeado para TFormTeste. Na **Listagem 8** está sendo apresentada a implementação do mesmo.

O método btnCalcularImpostosClick corresponde ao evento OnClick do botão btnCalcularImpostos, sendo que a partir do mesmo será utilizada a classe de fachada TImpostosFacade (variável facadeImpostos): busca-se com isto obter uma instância de TImpostosItem, a qual conterá as bases e os valores de impostos calculados para as informações digitadas por um usuário.

Listagem 8. Implementação do formulário TFormTeste

```
unit FormTestePatterns;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
 Dialogs, StdCtrls, ExtCtrls;

type
 TFormTeste = class(TForm)
   painelInformacoesItem: TPanel;
   painelImpostos: TPanel;
   btnCalcularImpostos: TButton;
   btnSair: TButton;
   lblValorItem: TLabel;
   txtValorItem: TEdit;
   lblPercReducaoICMS: TLabel;
   txtPercReducaoICMS: TEdit;
   lblAliquotaIPI: TLabel;
   txtAliquotaIPI: TEdit;
   lblAliquotaICMS: TLabel;
   lblVlBaseIPI: TLabel;
   txtVlBaseIPI: TEdit;
   lblVlIPI: TLabel;
   lblVlBaseICMS: TLabel;
   txtVlBaseICMS: TEdit;
   lblVlICMS: TLabel;
   txtVlICMS: TEdit;
   txtAliquotaICMS: TEdit;
   txtVlIPI: TEdit;
   procedure btnSairClick(Sender: TObject);
   procedure btnCalcularImpostosClick(Sender: TObject);
 private
   function FormatarValorImposto(valor: Real): string;
 end;

var
 FormTeste: TFormTeste;

implementation

uses ImpostosFacade, InformacoesImpostos;

{$R *.dfm}

function TFormTeste.FormatarValorImposto(valor: Real): string;
  var strValor: string;
   begin
   Str(valor:10:2, strValor);
   Result:= strValor;
end;

procedure TFormTeste.btnCalcularImpostosClick(Sender: TObject);
var facadeImpostos: TImpostosFacade;
   impostosItem: TImpostosItem;
begin
try
   facadeImpostos:= TImpostosFacade.Create;
   impostosItem:= facadeImpostos.CalcularImpostosItem(
   StrToFloat(txtValorItem.Text),
   StrToFloat(txtAliquotaICMS.Text),
   StrToFloat(txtPercReducaoICMS.Text),
   StrToFloat(txtAliquotaIPI.Text));

   txtVlBaseIPI.Text:= FormatarValorImposto(
   impostosItem.ValorBaseIPI);
   txtVlIPI.Text:= FormatarValorImposto(
   impostosItem.ValorIPI);
   txtVlBaseICMS.Text:= FormatarValorImposto(
   impostosItem.ValorBaseICMS);
   txtVlICMS.Text:= FormatarValorImposto(
   impostosItem.ValorICMS);

   ShowMessage('Processamento dos impostos efetuado com sucesso!');
   except
   on ex: Exception do
   ShowMessage('Erro durante o cálculo dos impostos: ' +
   ex.Message);
   end;
end;

procedure TFormTeste.btnSairClick(Sender: TObject);
begin
 Close;
end;

end.
```

A utilização do objeto TImpostosFacade torna mais do que aparente a vantagem de se utilizar o pattern Façade. Através desta prática evitou-se que o formulário precisasse instanciar uma série de objetos, bem como necessitasse coordenar uma série de chamadas a tais classes.

Executando a aplicação de testes

Ao se executar o projeto TesteStrategyFacade será exibida uma tela como a indicada na **Figura 3** na qual devem ser preenchidos os campos "Valor do Item", "% Alíquota ICMS", "% Red. Base ICMS" e "% Alíquota IPI".

Tão logo os campos que servem de base para os cálculos tenham sido preenchidos, deve-se executar a opção "Calcular Impostos" a fim de se obterem os resultados esperados. A **Figura 4** é um exemplo de teste que se efetuou por meio da aplicação demonstrada neste artigo.

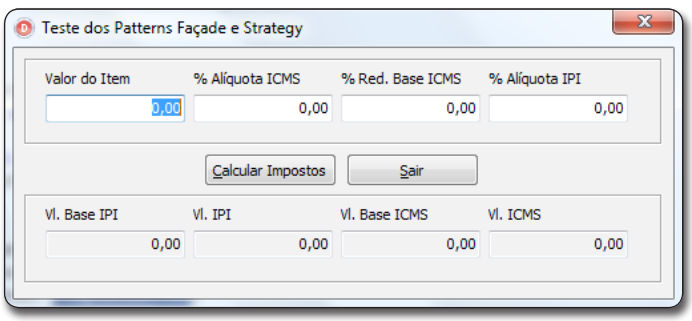


Figura 3. Tela inicial que é exibida ao se executar o projeto TesteStrategyFacade

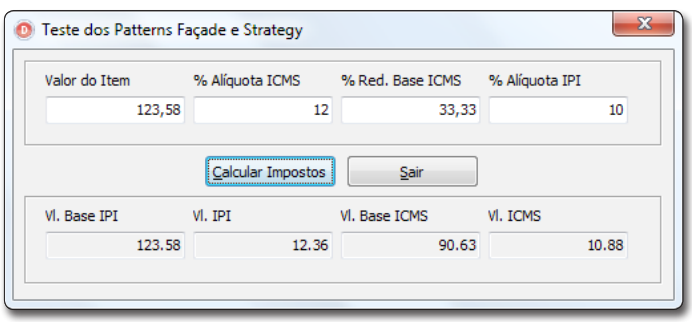


Figura 4. Exemplo de teste de cálculo de impostos

Conclusão

O uso de design patterns é um fator-chave para a obtenção de aplicações bem estruturadas, com os componentes destas últimas apresentando um bom grau de coesão e possuindo um baixo acoplamento entre si. Ao longo dos anos muitos padrões foram documentados por desenvolvedores de software, sendo que a adoção de práticas deste tipo pode não apenas melhorar a qualidade dos programas obtidos, bem como resultar em menor tempo de desenvolvimento e economia de recursos destinados aos projetos.

A utilização do pattern Strategy pode ser extremamente útil em situações nas quais um determinado aspecto está sujeito a uma série de variações. Já o padrão Façade pode simplificar operações complexas, fornecendo para isto um ponto que controla tais ações e expondo o acesso às mesmas de uma maneira mais sucinta.

Escolher quais patterns serão implementados em uma solução dependerá das características esperadas para esta última. Um estudo de prováveis padrões a serem empregados em um projeto pode contribuir não apenas para se chegar ao produto final da forma esperada, bem como facilitar ainda a evolução futura da aplicação.

Renato José Groffe
renato.groffe@yahoo.com.br
Atua como consultor em atividades voltadas ao desenvolvimento de softwares há mais de 10 anos. Bacharel em Sistemas de Informação, com especialização em Engenharia de Software. Microsoft Certified Technology Specialyst (Web, WCF, Distributed Applications, ADO.NET, Windows Forms), Oracle Certified Associate (PL/SQL), Sun Certified (SCJP, SCWCD), ITIL Foundation V2.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Criando uma deskbar

Aplique POO e refatoração

Resumo DevMan

De que trata o artigo:

O artigo trará uma abordagem prática dos temas programação orientada a objetos, generics, herança x composição, refactoring, interface com o usuário e API do Windows. Esses temas serão combinados de modo a produzir uma Deskbar que pode conter ícones para programas.

Em que situação o tema é útil:

O conteúdo apresentado neste artigo servirá para criar sistemas com visual mais atraente para os usuários, mais práticos de se usar e dando um acabamento mais "profissional" ao formulário principal do software. O tema é útil sempre que houver a necessidade de se inovar na apresentação ou comportamento do software. Criação de imagens ou ícones "arrastáveis" para outros programas do computador ou para menus e formulários do sistema. Também será apresentado como utilizar e agregar componentes existentes para se mesclar funções de um componente em outro sem reinventar a roda.

Criando uma deskbar:

Deskbars, também conhecidas como sidebars, toolbars e outras expressões que remetem à ideia de barra de ferramentas em sua área de trabalho, são ferramentas muito úteis para dar um visual agradável ao desktop do computador e manter certas funções sempre à mão, à distância de um clique. Uma ferramenta dessas deve conter atalhos para outras ferramentas e agregar algumas das funções mais usadas no PC como notificação de e-mails e relógio, listas de afazeres, leitores de RSS entre outras opções. Mas para ser produtiva a deskbar não deve atrapalhar o usuário ou se "intrometer" em suas tarefas quando isso não for necessário.

Existem no mercado vários programas para atuar como barra de ferramentas da área de trabalho. No Windows XP era possível colocar atalhos em uma pasta e "arrastar" esta pasta contendo somente atalhos para um canto do desktop, que ela se tornava automaticamente uma barra de ferramentas. Era possível configurar essa barra para se auto-ocultar e para exibir ícones pequenos e sem descrição, tornando a área de trabalho bem limpa, pois era possível se livrar de todos aqueles atalhos na área de trabalho.

No Windows 7 há aqueles gadgets da área de trabalho que podem conter um relógio, leitores de RSS e outras (in)utilidades. No entanto não é possível colocar atalhos para programas nesta barra.

VITOR LUIZ RUBIO

Outro produto digno de nota é o google desktop, que além de trazer várias funcionalidades para a desktop ainda conta com as buscas do google e com uma eficiente indexação e busca de e-mails e arquivos em seu próprio computador. Esta também não permite criação de atalhos para programas, mas possui uma função interessante onde é possível digitar o nome de um programa, e o mesmo será aberto, caso seja encontrado.

Há os programas que criam apenas barras de atalhos bonitas para tentar imitar o estilo das barras de ferramentas do Mac, como a RK Launcher e a Object Dock. Esses aplicativos podem parecer simples, mas há uma lógica bem elaborada em seu comportamento e na sua apresentação. Por exemplo o recurso de ocultar-se automaticamente faz com que a barra desapareça em uma lateral qualquer da tela para reaparecer apenas quando o mouse toca essa lateral. Uma desvantagem desse comportamento é que a barra pode ser ativada "sem querer" caso seja necessário levar o mouse para um ponto extremo da tela, como o botão fechar de uma janela maximizada ou para virar a "câmera" em um jogo encostando o cursor no limite da tela para "enxergar" do outro lado.

Se a barra estiver na direita ela pode ser ativada sem querer ao se clicar no botão fechar de um programa. Na parte inferior ela será ativada sempre que for necessário clicar no menu iniciar ou em uma janela minimizada. Na esquerda atrapalha sempre que se for usar os ícones da esquerda, como a lixeira, e na parte superior ela pode atrapalhar ao se navegar na internet ou acessar menus de janelas maximizadas.

Parece uma coisa simples, e que pode ser configurada, mas ninguém pensa nisso. Usabilidade também significa ter o comportamento que vai incomodar menos por padrão, sem a necessidade de se configurar nada. E principalmente para quem trabalha com o computador o dia inteiro, usabilidade significa produtividade. A produtividade é diferente para cada tipo de trabalho. Enquanto uma "notificação de e-mail" pode tornar um vendedor ou comprador super eficiente em responder seus e-mails e se comunicar ou negociar com clientes e fornecedores, a mesma notificação de e-mail pode atrapalhar um programador ou designer quando este estiver precisando de concentração.

Particularmente eu acho interessante o comportamento de ao se aproximar o mouse exibir apenas uma área bem fina da barra de ferramentas, e esta só abrir mediante um clique. Acho que esse comportamento não prejudica a usabilidade e não atrapalha no uso de um browser, por exemplo.

Nota do DevMan

Agregação é uma relação todo-parte um pouco mais fraca do que a composição. Por exemplo no caso do ícone montado, a partir de um TImage e de um TLabel tem-se uma agregação pois tanto TImage como TLabel fazem sentido separados e podem coexistir separadamente.

Composição é um tipo de associação mais rígida. Quando se tem uma composição os membros desta não fazem sentido separadamente e não podem existir fora do todo. O comportamento do todo é muito diferente do comportamento das partes, e as partes são destruídas caso o todo seja destruído.

Bons exemplos de agregação e composição se encontram na culinária. Por exemplo um omelete é uma agregação de ovos, queijo, presunto e outros ingredientes que você goste. No caso da omelete os componentes mantêm suas características, o queijo, o presunto etc. são bem perceptíveis e fazem sentido existindo fora da omelete. Já um bolo é um caso de composição, pois o bolo apresenta características e comportamento muito diferentes dos ingredientes isolados, farinha, ovos, açúcar e fermento. Os ingredientes do bolo não fazem muito sentido separados, por exemplo, ninguém come ovos com fermento. Em agregação o todo pode fazer sentido sem as partes, em composição não pode. Essa é a diferença entre agregação e composição.

Vale ressaltar que algumas dessas barras usam recursos gráficos como transparências e animações que as tornam pesadas e inviáveis para uso em netbooks, por exemplo.

Alguns conhecimentos teóricos serão apresentados neste artigo que podem ser úteis no desenvolvimento de outras aplicações. São eles:

Agregação → Não é só com herança que se estende as funcionalidades de uma classe. Se uma classe tiver internamente em seu corpo campos que são objetos de outras classes, ela pode incorporar as funções destes. Por exemplo, um ícone é uma figura com um texto embaixo, então pode-se pensar que um ícone pode ser "montado" com um TImage que tem um TLabel na parte inferior.

Generics → Generics facilitam muito a criação de listas tipadas dinâmicas. Como uma barra de ferramentas lateral pode ter várias listas de coisas (widgets, ícones etc.) listas genéricas podem ser usadas para atingir esse objetivo.

API do Windows → Arrastar um ícone ou arquivo para dentro de uma aplicação só é possível mediante o uso da API. Primeiro deve-se informar que determinada janela (ou panel, ou handle) pode receber arquivos arrastados. Depois é preciso dizer ao sistema o que fazer com os arquivos arrastados.

Ícones e menus na system tray → É importante para "travar" a barra ou fechá-la.

ClientDataset desconectado → A lista de "ícones", por exemplo, será gravada em um arquivo XML do ClientDataset sem o uso de providers ou bancos de dados.

Noções Gráficas → Conversões de ícone para bitmap, transparências, aparência.

Registro do Windows → A barra deve iniciar junto com o Windows, mas não deve ser um serviço pois serviços iniciam antes do login, e rodam mesmo se nenhum usuário se logar. No caso de uma barra de atalhos, ela é um programa essencialmente visual, por isso precisa iniciar apenas depois do login, para trazer apenas os atalhos do usuário logado.

Componentes Jedi → Baixe e instale o pacote de componentes JEDI, pois o componente JvBackGround será usado. Ele serve para colocar uma imagem como background de um formulário evitando flicking. Além disso esse componente tem propriedades interessantes como definir uma cor transparente e usar stretch, que significa esticar a imagem preenchendo todo o tamanho do formulário.

A **Figura 1** mostra como será a classe TIconeComando sem os métodos e propriedades, apenas o rascunho de sua interface. Uma lista de ícones é uma agregação de vários ícones, depende da classe TLabel pois tem um label interno e possuirá uma ligação com um menu popup para ações como excluir. Um TIconeComando é descendente do TImage.

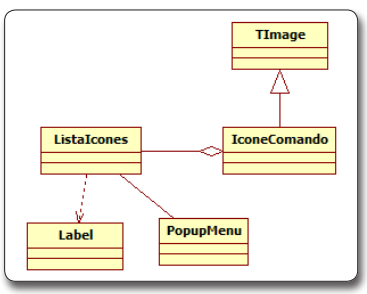


Figura 1. Diagrama de classes

Criando a "casca"

O primeiro passo é criar o comportamento. Uma barra de ferramentas deve ser um formulário "fino" que deve se acoplar a uma das laterais do seu desktop. O ideal é que o lado seja configurável, mas por questões de simplicidade este artigo assume que a barra ficará sempre do lado direito.

Ao iniciar, a barra não pode ter qualquer tipo de borda, muito menos botões de minimizar, maximizar ou fechar. Por isso é bom colocar BorderStyle := bsNone na inicialização.

Uma fina fatia do formulário deve ser sempre visível. Essa fatia deve ser larga o suficiente para disparar o evento OnMouseEnter quando o ponteiro do mouse passar por ela.

Passando o ponteiro por ela deve "crescer" vindo pouca coisa mais para a direita. Tirando o mouse dela, ela deve voltar à posição inicial. Todo esse comportamento cogitado até agora pode ser controlado manipulando-se apenas a propriedade left do formulário. Por exemplo, se o formulário tem 500 pixels de largura e a propriedade left for "TamanhoDaJanela – 20" isso significa que os 20 pixels da esquerda serão visíveis e os outros 480 estarão "para fora da tela". O tamanho da tela é possível de se obter com Screen.Width.

Para isso um formulário como o da **Figura 2** foi criado. Ele possui um ClientDataset, um TrayIcon, dois popup menus (um para o TrayIcon e outro será visto mais adiante) e um SpeedButton.

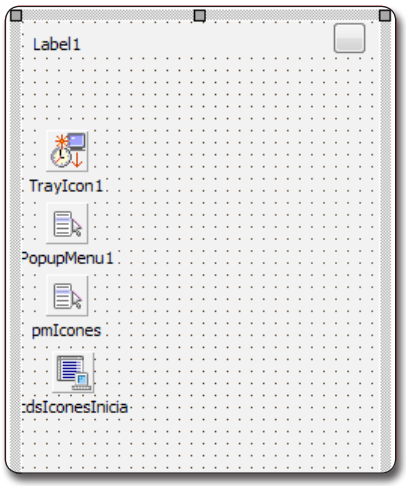


Figura 2. Formulário inicial

No create do formulário é executado o método Inicializa que tem por responsabilidade setar o tamanho e posição iniciais do formulário e deixar uma fina fatia visível à direita. Caso o leitor se interesse por configurar a posição onde a barra de ferramentas ficará então, caso a barra fique à esquerda, a lógica desse método deve ser invertida fazendo com que a parte visível seja um pequeno pedaço da direita do formulário, mostrado à esquerda da tela.

Outra responsabilidade deste método é inicializar o ClientDataset, criando o arquivo CDS caso o mesmo não exista. O arquivo CDS deve ser diferente para cada usuário do sistema, por isso cada usuário terá uma pasta com seu próprio arquivo CDS. A **Listagem 1** mostra como criar o arquivo CDS para cada usuário. O método NomeUsuario é um método estático, público e global que funciona como wrapper para o método GetUserName da API do Windows e retorna o nome do usuário logado no momento. No código fonte disponível pode-se ver o método completo.

O método PosicaoInicial retorna o tamanho da tela subtraindo-se dez pixels. Isso quer dizer que a propriedade left do formulário será dez pixels antes do limite direito da tela, fazendo com que uma fina camada dessa largura seja exibida.

Além de chamar o método Inicializa na criação do formulário deve-se configurar os eventos OnMouseEnter, OnMouseLeave e OnClick do formulário. Ao se clicar o formulário deve ser totalmente exibido. Ao se sair ele deve voltar para a posição inicial caso não esteja travado. Ao se entrar deve mostrar um pouco mais do que está oculto sem mostrar tudo. Esse comportamento é desejado para que a barra não atrapalhe programas que funcionem em tela cheia e jogos. O evento OnMouseEnter deve chamar o método SegundaPosicao, como é mostrado a seguir:

```
procedure TfrmSideBar.SegundaPosicao;
begin
 if not (FTravado or FAberta) then
  Left := Screen.Width - 20;
end;
```

Listagem 1. Inicialização do formulario

```
procedure TfrmSideBar.Inicializa;
var
 NomeArquivo: string;
 DiretorioArquivo: string;
begin

 DiretorioArquivo := ExtractFilePath( ParamStr(0) )+'\'+ NomeUsuario+'\';
 NomeArquivo :=  DiretorioArquivo + 'icones.cds';
 if not DirectoryExists(DiretorioArquivo) then
  ForceDirectories(DiretorioArquivo);
 cdsIconesInicia.FileName := NomeArquivo;
 if not FileExists(NomeArquivo) then
  cdsIconesInicia.CreateDataSet;
 cdsIconesInicia.Open;

 Left := PosicaoInicial;
 Top := 0;
 Height := Screen.Height;
 FAberta := false;
 FTravado := false;

end;

function TfrmSideBar.PosicaoInicial: integer;
begin
 Result := Screen.Width -10;
end;
```

Os campos privados FTravado e FAberta são do tipo boolean e indicam se a barra está travada e se ela está sendo mostrada, respectivamente. A barra estar travada indica que ela permanecerá sendo exibida não importa se o mouse entre ou saia dela. Para isso foi colocado um BitBtn no canto superior direito que tem como código: FTravado := not FTravado;

O método da **Listagem 2** deve ser executado no evento OnMouseLeave e será disparado sempre que a seta do mouse sair da área da form.

O método MouseDentro identifica se o cursor do mouse está dentro da área do formulário. Caso a barra funcione de múltiplos lados do desktop a lógica desse método deve ser modificada. Isso é necessário pois o evento OnMouseLeave é executado quando se entra em um outro controle dentro do próprio formulário, como um botão ou painel. Isso é indesejável nesse contexto, pois com o formulário sendo exibido não é interessante que o método OnMouseLeave seja executado ao se passar o mouse por um botão ou painel na barra. Isso faria com que a barra voltasse ao tamanho inicial prematuramente.

O método MouseDentro é simples, baseado na posição do cursor em relação à posição do formulário, e pode ser visto na sequência:

```
function TfrmSideBar.MouseDentro: boolean;
begin
 Result := (Mouse.CursorPos.X >= left) and (Mouse.CursorPos.Y >= Top);
end;
```

Listagem 2. Método VoltarPosicaoInicial

```
procedure TfrmSideBar.VoltarPosicaoInicial;
begin

 if MouseDentro then
  exit;

 if  not (FTravado) then
 begin
  Left := PosicaoInicial;
  FAberta := false;
 end;

end;
```

Ao se clicar no formulário ele deve assumir a posição que mostre todo o seu conteúdo. Isso é assegurado pelo método TamanhoAberto exibido a seguir:

```
procedure TfrmSideBar.TamanhoAberto;
begin
 Left := Screen.Width - Width;
 FAberta := true;
end;
```

Com isso o comportamento básico do formulário já foi definido. Mais adiante novos métodos serão adicionados para lidar com os ícones e depois o comportamento do formulário será desacoplado por meio de um refactoring. Mas primeiro os ícones devem ser definidos.

Nesse ponto já é possível testar o comportamento da barra lateral. O comportamento esperado deve ser mostrar uma pequena área ao iniciar, expandir essa área ao passar o mouse, mostrar tudo ao se clicar, esconder ao se tirar o mouse ou continuar mostrando caso se clique no botão "travar". O código desse botão travar tem apenas uma linha, mas será modificado mais adiante:

```
FTravado := not FTravado;
```

Uma barra de ferramentas desse tipo não deve mostrar seu ícone na barra de tarefas do Windows, por isso no arquivo DPR a propriedade Application.MainFormOnTaskbar é definida como False, assim o handle da janela/ícone na barra de tarefas não é o mesmo handle da form principal e sim um handle próprio, o handle de Application.

Nota

Nas versões anteriores do Delphi não existia a propriedade Application.MainFormOnTaskbar, e a janela na barra de tarefas era sempre a Application. Isso é uma das propriedades que foram adicionadas nas novas versões do Delphi. Quando essa propriedade é false o handle do ícone na barra de tarefas é o mesmo que o de application, como tradicionalmente, e o caption que aparece lá é o nome da aplicação. Já quando essa propriedade é true o handle do ícone na barra de tarefas é o mesmo da form principal, assim como seu caption.

No evento OnActivate do formulário deve ser executado o seguinte comando:

```
ShowWindow(Application.Handle, SW_HIDE);
```

Isto para que o ícone na barra de tarefas seja escondido. Se a propriedade Application.MainFormOnTaskbar fosse true a linha ShowWindow(Application.Handle, SW_HIDE) não teria efeito algum.

Criando os ícones

Neste projeto os ícones serão descendentes de TImage, um controle feito para se mostrar imagens em um formulário. Para isso será definida uma classe que descenda de TImage e ela possuirá a lógica necessária para extrair o ícone de um programa e exibi-lo, e executar um programa quando clicado.

Algumas decisões de arquitetura devem ser tomadas neste ponto. Por exemplo se os ícones serão aproveitáveis em outros sistemas ou não, se serão arrastáveis, selecionáveis, se abrirão com um clique ou com dois cliques, se terão algum tipo de destaque ao se passar o mouse e assim por diante.

Identificar e programar comportamentos específicos para clique, duplo-clique, selecionar e assim por diante pode ser útil caso esteja programando um sistema que tenha uma "desktop". No final deste artigo serão mostradas algumas alterações que podem ser feitas nesta classe para que se comportem como ícones de um desktop. A vantagem disso é que é possível criar sistemas cuja janela principal tenha ícones para programas usados frequentemente ou para funções do próprio sistema.

O primeiro passo é criar uma classe que seja descendente de TImage. A esta classe devem ser adicionados propriedades e métodos para que se comporte como o desejado. Esta classe será chamada de TIconeComando, filha de TImage. TImage é o componente que permite colocar imagens em formulários ou painéis, e se encontra na paleta Additional.

Cada ícone terá uma referência a uma lista de ícones. Como a referência à lista de ícones está na interface de TIconeComando e a própria lista tem uma referência a esta classe então ambas devem ser declaradas na mesma unit, pois não é possível units com referências circulares na seção interface, apenas na seção implementation.

Nota

Referência circular é quando uma unit A precisa de algo na unit B e a unit B precisa de algo na unit A. Na seção implementation, que é o uses "de baixo", uma unit pode usar recursos da outra e elas podem ter referências circulares, mas na seção interface, que é o uses "de cima", as units não podem ter referências circulares.

A classe que implementa a lista de ícones chama-se TIconeList. Ela terá uma lista de TIconeComando, e cada TIconeComando da lista terá uma referência a TIconeList onde ele estiver. A TIconeList será uma lista genérica, mas terá alguns métodos a mais. Uma de suas responsabilidades é sobrecarregar o método Add para que cada ícone adicionado à lista faça uma ligação com a própria lista onde está sendo inserido.

Outras responsabilidades da lista podem incluir desfazer a seleção dos outros ícones quando um deles está selecionado (supondo que o leitor queira fazer ícones selecionáveis), ou organizando-os em sua superfície, tomando o cuidado para que não se sobreponham.

Para que as classes se referenciem uma à outra é necessário uma declaração prévia da lista.

Nota

Declaração prévia é quando uma classe A depende de uma classe B, mas esta classe B também depende da classe A, então antes da declaração da classe A declara-se a classe B vazia e sem corpo. No caso desse exemplo a classe TIconeList é declarada antes da TIconeComando apenas com seu tipo básico e sem corpo, e redeclarada após TIconeComando de maneira completa.

A **Listagem 3** mostra apenas a declaração das classes. A implementação de cada método será explicada adiante.

Como pode ser visto, a Listagem também mostra o uses, pois aqui são necessárias várias units para o correto funcionamento dos ícones. Começando pela shellapi, pois um ícone quando clicado executa um programa através do método ShellExecute. As units Graphics e PngImage servem para se manipular imagens. A unit Generics.Collections serve para se criar listas genéricas.

Repare que TIconeList está pré-declarado como sendo apenas class, mas é redeclarado depois. Isso é necessário para que TIconeComando possua uma referência a TIconeList.

As duas constantes ICONELEFT = 25 e TAMANHOICONE = 35 definem o tamanho de um ícone: um quadrado de 35 x 35, e a distância padrão da borda esquerda de seu formulário ou contêiner. Repare que TAMANHOICONE não é o tamanho real do ícone, que tem um tamanho inicial de 32, mas sim o tamanho do ícone somado a uma certa folga, para os ícones não ficarem encostados uns nos outros.

Logo a seguir há a declaração de TIconeComando como sendo TIconeComando = class(TImage). Os campos privados são o diferencial entre um ícone de comando e um TImage Comum. FComando armazena o endereço de um programa ou arquivo que será aberto. Numa aplicação maior ele poderia guardar o nome de

Nota do DevMan

Acoplamento alto é quando duas classes dependem muito uma da outra, ou quando as interfaces dependem muito da implementação, sendo que uma delas não pode ser substituída.

Com a referência circular criada neste artigo entre a lista e os ícones criamos um acoplamento. Isto pode ser uma característica ruim, um sinal de projeto fraco ou de alguma coisa que poderia melhorar.

É possível reduzir esse acoplamento pela criação de uma terceira classe que controle tanto a lista como os ícones.

Nota do DevMan

O Shell de um sistema operacional é a parte mais externa do sistema, com a qual o usuário interage e que pode interpretar comandos, sejam eles escritos, gestuais ou eventos de mouse e teclado. No caso do Windows o Shell é formado principalmente pelo Explorer.exe e pelo cmd.exe. Esses são os componentes "visíveis" aos usuários. Há ainda uma série de componentes menores e DLLs.

um menu ou botão a ser executado, ou até mesmo o nome de um método de um objeto. Na última parte deste artigo serão vistos exemplos mais bem elaborados assim, por enquanto Fcomando apenas dá comandos para o shell do Windows.

Nota

Se preferir também é possível separar descrição e hint em duas propriedades diferentes, assim o texto do label e o texto dos hints poderiam ser diferentes.

O campo FLista é a referência à instância de TIconeList onde este ícone está contido. FLabel é um TLabel usado para mostrar a descrição ou nome do comando logo abaixo do ícone (tendo em mente que um ícone é uma imagem arrastável com um texto abaixo).

O campo FPressionado indica se o ícone foi pressionado com o evento MouseDown do mouse. Caso esteja sendo pressionado ele pode ser arrastado.

O FClicado será verdadeiro sempre que ocorrer um clique. Um clique é um conjunto sequencial de MouseDown e MouseUp, ou seja, sempre que o botão esquerdo do mouse é pressionado e logo depois solto em uma mesma posição, ou em um mesmo ícone.

A função GetDescricao é o método acessor "Get" da propriedade Descrição. Ela usa o caption do FLabel. As funções GetTop e

Listagem 3. Declaração das classes e uses

```
unit Icones;

interface

uses
  Windows,
  Classes,
  SysUtils,
  ExtCtrls,
  Controls,
  ShellAPI,
  Dialogs,
  forms,
  menus,
  Graphics,
  pngimage,
  Contnrs,
  StdCtrls,
  ImgList,
  Generics.Collections;


const
  ICONELEFT = 25;
  TAMANHOICONE = 35;

type

  TIconeList = class; //Declaração prévia, para uso dentro desta mesma unit

  TIconeComando = class(TImage)
  private
    FComando: string;
    FLista: TIconeList;
    FLabel: TLabel;         //texto abaixo do ícone e descrição/hint
    FPressionado: Boolean;    //define se o mouse foi presionado antes do drag
    FClicado: Boolean;      //define se o objeto foi clicado/marcado/selecionado

    //gets
    function GetDescricao: string;
    function GetTop: Integer;
    function GetLeft: Integer;
    //sets
    procedure SetDescricao(vdescricao: string);
    procedure SetComando(vComando: string);
    procedure SetTop(vTop: Integer);
    procedure SetLeft(vleft: Integer);
    procedure SetTopLabel(const vTop: integer);

    //eventos
    procedure DoDblClick(sender: TObject);
    procedure DoMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure DoMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure DoMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
    procedure DoClique(Sender: TObject);
    procedure DoMouseEnter(Sender: TObject);
    procedure DoMouseLeave(Sender: TObject);

    //procedures / function
    procedure GrudaLabel;
    procedure SetaImagem;
    procedure LoadBitmapFromFile(filename: string);
    procedure ConvertBitmapFromIcon(filename: string);
    procedure LoadBitmapFromIcon(filename: string);
    procedure SetTamanho;
    function PosicRelat(X, Y: Integer): TPoint;
    function ParentCanvas: TCanvas;

    procedure SetSelected;
    procedure SetUnselected;
    procedure Marca;
    procedure Desmarca;

  protected
    procedure SetParent(vParent: TWinControl); override;

  public
    constructor Create(aowner: TComponent); override;
    destructor Destroy; override;
    property Lista: TIconeList read FLista write FLista;
    property Descricao: string read GetDescricao write SetDescricao;
    property Comando: string read FComando write SetComando;
    property Top: Integer read GetTop write SetTop;
    property Left: integer read GetLeft write SetLeft;
  end;

  //precisa ser criado. O genérico  TObjectList<TIconeComando> não pode ser
  //usado devido às novas propriedades que serão criadas
  //como a dependência de TIconeList em relação a TIconeComando e a
  //TIconeComando em relação a TIconeList são ambas na interface isso criaria
  //uma referencia circular no uses.
  TIconeList = class(TObjectList<TIconeComando>)
  private
    FItemIndex: Integer;
  protected
    procedure SetItemIndex(vIndex: Integer); virtual;
    function GetItemIndex: Integer; virtual;
  published
    function Add(vIcone: TIconeComando): integer; {Insere o icone no final da coleção}
    property ItemIndex: Integer read GetItemIndex write SetItemIndex;
  end;
```

GetLeft são apenas os acessores Get das propriedades Top e Left, que executam os métodos homônimos do ancestral TImage. Eles estão aí apenas porque o SetTop e o SetLeft serão diferentes: eles têm que encaixar o label logo abaixo da imagem sempre que a imagem é movida.

O SetDescricao é o método modificador "Set" da propriedade Descrição. Além de preencher o caption do FLabel ela preenche o hint do FLabel. Assim, passando o ponteiro do mouse por cima da imagem ou do label aparecerá um hint com o mesmo texto.

O SetTopLabel é executado junto com os novos SetLeft e SetTop e serve para "colar" o label ao pé da imagem.

Em seguida há uma série de métodos com assinatura: procedure xyz(sender: Tobject). Esses métodos privados servem para definir os eventos clique, duplo clique, entrar com o mouse, sair com o mouse e assim por diante. Eles são atribuídos aos eventos na criação do TIconeComando. Essa é a maneira mais didática de criar esses eventos padrão, mas não impede que esses eventos sejam sobrescritos ou invalidados via programação. Para melhorar isso é possível criar esse comportamento sobrecarregando os métodos MouseDown, MouseUp, DblClick, Click, MouseMove e assim por diante. Todos esses métodos são virtuais, o que significa que podem ser sobrecarregados garantindo o polimorfismo, e todos eles são protected, o que garante que podem ser visíveis, executados e sobrecarregados pelos seus descendentes.

Outro comportamento que se deve ter em mente é como os ícones serão organizados, se poderão ter uma ou mais colunas, qual o tamanho máximo vertical e qual o tamanho máximo horizontal, além dos espaçamentos entre eles. Esse comportamento pode ser adicionado no formulário que conterá os ícones, mas assim eles não podem ser organizados automaticamente.

Para que os ícones sejam organizados automaticamente sempre que um novo ícone é adicionado ou arrastado o método OrganizarIcones deve ser um método do ícone ou da lista de ícones, ou então de uma terceira classe controladora.

Não faz sentido imaginar que um ícone deveria organizar todos os outros, mas ele pode solicitar isso à lista. Já a lista deveria conhecer os limites horizontal e vertical para não ultrapassá-los.

É possível organizar os ícones pela ordem em que foram adicionados na lista ou pela ordem em que estão gravados no ClientDataset. Também é possível organizar baseado nas posições relativas, ou seja, pelo seu valor top, apenas adicionando ou subtraindo unidades para que eles fiquem com o mesmo espaçamento, equivalente a um "Alinhar à Grade" de qualquer aplicação em que se movimentem objetos por uma superfície.

A maneira como isso foi implementado será vista mais adiante. Aqui fica evidente um possível refactoring: passar o dataset para dentro da lista de ícones ou qualquer classe que os controle.

O método GrudaLabel serve para sempre juntar o label à parte inferior do ícone, mesmo ele sendo movido. Ele dá um espaço de um pixel como respiro, para não ficar totalmente encostado no TImage.

O método SetaImagem é apenas um wrapper, ele tem por obrigação chamar o método apropriado para carregar o ícone ou imagem. Isso é necessário porque pode ter mais de um método para carregar a imagem, por exemplo a imagem pode ser carregada a partir do ícone associado ao arquivo ou aplicativo, usando a API do Windows, pode ser carregado dessa forma mas convertido para BMP ou PNG ou ainda pode simplesmente ser carregado de uma imagem BPM ou PNG no disco rígido.

Um desses métodos é o LoadBitmapFromIcon, outro é o ConvertBitmapFromIcon e o terceiro é o LoadBitmapFromFile, que serão vistos mais adiante.

A função PosicRelat retorna um objeto do tipo TPoint, ou seja, um "ponto" num plano bidimensional com suas coordenadas X e Y. O objetivo dessa função é, dadas as coordenadas X e Y de um evento de movimentação ou deslocamento do mouse, somar essas coordenadas com as propriedades Left e Top atuais do ícone subtraídas da metade de seu tamanho. Isso faz com que o centro do ícone seja posicionado no ponteiro do mouse na hora de arrastá-los. Caso não se use essa função os ícones seriam sempre arrastados pela "ponta" e apareceriam arrastados no "canto" do formulário em vez de suas posições originais. A função também tem uma proteção caso os ícones sejam arrastados além dos limites visuais da barra de ferramentas.

A função ParentCanvas retorna o Canvas do controle que for o parent dos ícones. Isso é necessário para que se permita desenhar na superfície de formulários, painéis e frames. O canvas é como se fosse uma folha em branco que todos os controles do Windows têm e é usado para se desenhar gráficos, linhas, formas geométricas e figuras. A maneira como a VCL do Delphi encapsula o conceito do Canvas torna o seu uso muito fácil e transparente. No caso desse artigo o canvas é usado para desenhar e redesenhar o ícone pelo formulário quando ele é arrastado. A propriedade DoubleBuffered do formulário setada para True permite que nos bastidores os gráficos do formulário tenham um buffer a mais para evitar ou diminuir o flickering, que é o nome que se dá às piscadas que um formulário dá quando se arrasta um componente gráfico por ele.

Os métodos SetSelected e SetUnselected mudam as cores do label e da fonte do label, além da quebra de linha do conteúdo do label, além de atribuir os valores True ou False para o campo FClicado. Esses métodos cuidam apenas da parte gráfica e lógica dos ícones. Os métodos Marca e Desmarca executam o SetSelected e SetUnselected, respectivamente, ao mesmo tempo em que setam qual item da lista está selecionado através da propriedade ItemIndex. O método Marca ainda tem a responsabilidade de, quando um ícone for selecionado, "desselecionar" todos os outros da lista.

Esses métodos foram divididos assim para facilitar alterações futuras e seu uso não é obrigatório. Caso o desenvolvedor decida que seus ícones não serão selecionáveis basta modificar, retirar ou restringir a execução desses métodos.

O método SetParent sobrecarrega o SetParent ancestral e tem por responsabilidade atribuir ao parent do label o mesmo parent do ícone. Assim, se o formulário da barra de ferramentas é o "parent" do ícone ele será o parent do label também.

O construtor instancia todos os objetos que fazem parte da composição, como o label, por exemplo, e atribui os eventos. O destructor faz o caminho inverso liberando a memória.

Já a lista de ícones não é apenas uma lista comum. Além de ser baseada em generics, o que facilita bastante o seu uso com tipos definidos de dados sem usar typecast, ela tem métodos próprios que adicionam comportamento. O método OrganizarIcones, já mencionado, o método Add, que adiciona um item à lista mas "linka" a lista com o item e os métodos SetItemIndex e GetItemIndex, acessores da propriedade ItemIndex, que analogamente a um listbox ou combo, marca qual item está selecionado.

O constructor, na **Listagem 4**, inicializa todos os campos, tamanhos, valores e instancia os objetos.

Os outros métodos podem ser conferidos no código fonte disponível para download. Alguns merecem destaque. O evento DoDblClick apenas seta FPressionado para False, já que se houve um duplo-clique em um ícone qualquer significa que ele não está sendo continuamente pressionado.

O evento DoMouseDown marca pressionado como true, clicado como true e executa o método Marca tornando assim o ícone selecionado. Os métodos DoMouseEnter e DoMouseLeave servem para destacar um ícone quando se passa o mouse por ele. Uma das maneiras de se fazer isso é mudar o tamanho do ícone, outra maneira, que será vista no próximo artigo, é usar algum outro componente que possa mudar sua própria cor, como um panel, para dar um efeito visual. O evento DoMouseUp verifica primeiro se o mouse estava sendo pressionado. Em caso positivo, significa que o ícone foi arrastado e está sendo deixado em uma determinada posição. O evento DoMouseMove executa um repaint no parent para apagar quaisquer rastros de uma pintura anterior do ícone, depois pinta o ícone na posição que o mouse está passando com o comando ParentCanvas.Draw(ponto.X, ponto.Y, self.Picture.Graphic, 128). Sem o repaint o ícone deixaria um rastro no formulário ao ser arrastado. O comando Draw do canvas desenha, numa coordenada X e Y, um bitmap contido em um objeto TGraphic. O último parâmetro é o grau de opacidade/transparência do desenho.

Nota do DevMan

Há muitas maneiras de se implementar o recurso de arrastar e soltar. A VCL do Delphi traz os métodos OnDragDrop, OnDragOver e métodos como StartDrag e EndDrag para muitos componentes. O uso desses eventos e métodos é recomendável quando se usa algo bem definido, pré existente, que deve ser arrastado por um formulário ou painel também pré-definido, mas não é obrigatório que se faça dessa maneira.

No caso de ícones em um formulário não se sabe quantos são, todos tem que ter o mesmo comportamento e não se tem acesso prévio ao formulário que os contêm. Uma das maneiras de se fazer isso é com o canvas, repintando o ícone na nova posição a cada movimento X Y do mouse.

Outra é usar um TImageList e definir uma imagem de "DragImage", que dá um efeito profissional, com semi-transparência no mesmo estilo que a Desktop do Windows. Um artigo inteiro poderia ser escrito só sobre drag & drop, esse estilo citado aqui será apresentado no próximo artigo.

No SetComando o SetaImagem é executado pois o ícone muda caso o comando mudar. Não que o usuário possa mudar isso, mas caso se mude o caminho de um comando via programação e ele passe a apontar para outro arquivo.

Listagem 4. Constructor e Destructor

```
constructor TIconeComando.Create(aowner: TComponent);
begin
  inherited create(aowner);
  FLabel := TLabel.Create(aowner);
  FClicado := False;
  Self.ShowHint := True;
  Self.Proportional := True;
  Self.Stretch := True;
  Self.Transparent := true;
  Width := 32;
  Height := 32;
  Self.Top := 1;
  Self.Left := 1;
  self.OnClick := DoClique;
  Self.OnDblClick := DoDblClick;
  Self.OnMouseDown := DoMouseDown;
  Self.OnMouseUp := DoMouseUp;
  Self.OnMouseMove := DoMouseMove;
  Self.OnMouseEnter := Domouseenter;
  Self.OnMouseLeave := Domouseleave;
  //label
  FLabel.ShowHint := True;
  FLabel.Color := clHighlight;
  FLabel.Transparent := True;
  FLabel.WordWrap := True;
  FLabel.AutoSize := True;
  FLabel.Constraints.MaxHeight := 40;
  FLabel.Constraints.MaxWidth := Self.Width * 2;
  FLabel.Width := Self.Width;
  FLabel.Top := Self.Top + self.Height;
  FLabel.Left := Self.Left;
  FLabel.OnClick := DoClique;
  FLabel.OnDblClick := DoDblClick;
  FLabel.OnMouseDown := DoMouseDown;
  FLabel.OnMouseUp := DoMouseUp;
  FLabel.OnMouseMove := DoMouseMove;
end;

destructor TIconeComando.Destroy;
begin
  inherited Destroy;
  FLabel.Free;
  FLabel := nil;
end;
```

O método DoClique também é um evento. Ele executará um ShellExecute do FComando e rodará um programa. O parâmetro ou constante SW_SHOWNORMAL tem o valor de 1 e serve para mostrar normalmente a janela do programa a ser executado. Outros valores dessa constante podem ser encontrados na unit Windows e podem fazer a janela iniciar minimizada, maximizada ou invisível.

Caso o programador deseje, essa execução do método ShellExecute pode ser chamada dentro de DoDblClick em vez de DoClique, assim o programa abriria com um duplo clique, como na desktop. Isso pode fazer sentido em sistemas com ícones na janela principal que emulem o desktop, mas não faz sentido em uma deskbar. Numa deskbar os programas devem abrir com um único clique.

O método LoadBitmapFromIcon usa a API ExtractAssociatedIcon para extrair o ícone associado a um programa e associá-lo com o TImage. Caso não seja encontrado o programa ou arquivo a ser executado é adicionado o ícone da própria aplicação.

ConvertBitmapFromIcon é uma alternativa a LoadBitmapFromIcon que primeiro carrega um ícone usando o mesmo processo que LoadBitmapFromIcon, mas depois cria um bitmap para converter este ícone. Depois de convertido para bitmap outro bitmap é criado para que se amplie o original. Para isso é usado o método stretchdraw do canvas, que tem como característica receber uma coordenada X Y inicial e um tamanho, e fazer um desenho "esticado" a partir desta coordenada, com o tamanho definido.

Antes de "esticar" a figura é definida uma cor que será transparente. Com essa cor é preenchido todo o fundo do bitmap antes de pintar a figura expandida por cima. A cor escolhida é $f600ff, um tom de pink pouco usado, que será usado novamente mais tarde. A ideia é que tudo que for dessa cor fique transparente no final.

Depois disso o bmp é transformado em png com cor transparente. Os objetos ícone, bmps e png são destruídos logo depois que a imagem é atribuída ao bitmap de picture com o método Assign.

O método LoadBitmapFromIcon traz um ícone pequeno, de 32 por 32 pixels, e não pode ser ampliado pois é associado diretamente ao handle de Picture.Icon.Handle e é um recurso de tamanho fixo controlado direto pela API do Windows. As propriedades stretch e proportional não surtem efeito sobre ícones. Embora o tamanho do ícone seja fixo a definição da imagem é melhor. Nesse contexto não é recomendável usar código para aumentar ou diminuir o tamanho do ícone como destaque ao se passar ou retirar o mouse, como visto nos métodos DoMouseEnter e DoMouseLeave.

Já no caso do método ConvertBitmapFromIcon, este cria uma imagem de 64 por 64, mas a encolhe proporcionalmente para 32 por 32 pixels. Usando esse método faz sentido usar eventos que ampliam o ícone no DoMouseEnter e DoMouseLeave.

O uso desses mecanismos de ampliação usando LoadBitmapFromIcon faz com que o label fique afastado da imagem, que permanece do mesmo tamanho. Não é um efeito agradável.

O método LoadBitmapFromFile usa um arquivo bmp para o ícone. Isso pode ser feito para um conjunto de ícones de programas muito comuns, que todos os usuários tenham, usando imagens no diretório do programa. Exemplo de programas que podem ser usados assim são o Explorer, o Internet Explorer e a calculadora. Com isso é possível ter imagens grandes e bem definidas para um conjunto de programas.

O método OrganizaIcones procura não deixar que dois ícones sejam sobrepostos, ou seja, na mesma posição X e Y. Para isso ele verifica se as posições de um ícone e do próximo são iguais e adiciona uma unidade de 40 pixels para frente ou para trás.

Um algoritmo real de "alinhamento à grade" é mais complexo que isso e envolve ordenar os ícones pelas suas posições relativas em vez de ordem alfabética ou ordem em que estão armazenados na lista.

Para que sejam alinhados ou organizados automaticamente sempre que um ícone é solto é necessário que se execute OrganizaIcones sempre no final de DoMouseUp, caso contrário pode-se omitir a chamada a esse método. Uma versão alternativa do OrganizaIcones, baseado na ordem da lista, pode ser visto na **Listagem 5**.

Listagem 5. Versão alternativa do método OrganizaIcones

```
procedure TIconeList.OrganizaIcones;
const
  TamIcone = 50;
  DisTLabelVer = 40;
var
  i, ver: integer;
  DistTotVer, DistTotHor: Integer;
begin
  ver := 1;

  DistTotVer := TamIcone + DisTLabelVer;

  for i := 0 to Self.Count - 1 do
  begin
    Self.Items[i].Top := ver;
    Inc(ver, DistTotVer);
  end;
end;
```

Adicionando e controlando ícones

Depois que as classes TIconeComando e TIconeList estiverem criadas é necessário fazer a ligação entre o formulário e os ícones, embutindo comportamento para receber e gravar ícones.

Primeiro de tudo é preciso criar três campos no ClientDataset: Vertical (integer), comando (string) e descrição (string). Os campos comando e descrição são autoexplicativos: enquanto o comando contém a string que é executada pelo ShellExecute a descrição contém o texto do label. Já o campo Vertical armazena a posição vertical (propriedade top) dos ícones. Se desejar controlar a posição horizontal também deve-se adicionar mais um campo "Horizontal". Com um clique de direita do mouse no ClientDataset escolha a opção "Create Dataset". Isso criará um ClientDataset desvinculado de providers e de bancos de dados capaz de manipular arquivos XML ou CDS (MyBase).

Em seguida é necessário usar a API do Windows para permitir que o formulário aceite arquivos arrastados para dentro dele. Para isso foi adicionado o método InicializaIcones, conforme visto na **Listagem 6**.

Esse método é chamado no evento OnFormShow, e a primeira coisa que ele faz é invocar a API DragAcceptFiles. Essa função recebe como parâmetro o handle da janela que "aceitará" os ícones. Não são apenas formulários que têm handles, mas vários outros controles dependentes da API do Windows que são considerados por ela como janelas, por exemplo frames e panels.

O método InicializaIcones também cria a instância da lista de ícones e abre o ClientDataset. Se já houverem dados no cds, por exemplo por causa de ícones adicionados em uma execução anterior, esses serão adicionados na lista.

O método FinalizaIcones fecha o dataset, abre novamente procurando cada um dos ícones e atualizando suas posições, já que as mesmas podem ser trocadas durante a execução da barra. Ele pode ser visto na **Listagem 7**.

O FinalizaIcones é executado no evento OnClose do formulário e executa logo no final o LiberaIcones. O LiberaIcones destrói a lista, destruindo junto todas as imagens. O AbreIcone chamado dentro do FinalizaIcones serve para procurar um ícone pelo seu

comando. Sendo assim não pode haver dois atalhos para um mesmo programa, comando ou função.

Listagem 6. Método InicializaIcones da deskbar

```
procedure TfrmSideBar.InicializaIcones;
var
 IconeTemporario: TIconeComando;
begin
 DragAcceptFiles(Handle, True);
 FlistaIcones := TIconeList.Create;
 cdsIconesInicia.Open;
 cdsIconesInicia.First;
 while not cdsIconesInicia.Eof do
 begin
  IconeTemporario := TIconeComando.create(nil);
  with IconeTemporario do
  begin
   Comando := cdsIconesIniciaCOMANDO.AsString;
   Descricao := cdsIconesIniciaDESCRICAO.AsString;
   Top := cdsIconesIniciaVERTICAL.AsInteger;
   Left := ICONELEFT;
   Parent := Self;
   PopupMenu := pmIcones;
  end;
  FlistaIcones.Add(IconeTemporario);
  cdsIconesInicia.Next;
 end;
end;
```

Listagem 7. Método FinalizaIcones

```
procedure TfrmSideBar.FinalizaIcones;
var
 i: Integer;
begin
 cdsIconesInicia.Close;
 try
  try
   for i := 0 to FlistaIcones.Count - 1 do
   begin
    if AbreIcone(FlistaIcones.Items[i].Comando) then
    begin
     cdsIconesInicia.Edit;
     cdsIconesIniciaVERTICAL.AsInteger := FlistaIcones.Items[i].Top;
     cdsIconesInicia.Post;
    end;
   end;
  except
  end;
 finally
  LiberaIcones;
 end;
end;
```

O método ExcluiIcones é chamado através de um popup menu (o menu dos ícones) e tem como objetivo excluir os ícones tanto da lista como do dataset, conforme mostrado no código a seguir:

```
procedure TfrmSideBar.ExcluiIcones;
begin
 if AbreIcone(FlistaIcones.Items[FlistaIcones.ItemIndex].Comando) then
 begin
  cdsIconesInicia.Delete;
  FlistaIcones.Delete(FlistaIcones.ItemIndex);
 end;
end;
```

Nota

Se você estiver desenvolvendo um sistema no qual um usuário deve abrir ou anexar um arquivo você pode criar algo como o WMDropFiles desse exemplo, assim o usuário teria que arrastar o arquivo para dentro do programa em vez de usar um OpenDialog convencional. Isso é usado por exemplo no Outlook, Outlook Express ou Thunderbird na hora de anexar arquivos em um e-mail: é possível arrastar o arquivo para dentro do programa e ele será anexado.

ExcluiIcones também executa internamente AbreIcone, e deleta o ícone pelo comando. Com o formulário aceitando arquivos arrastados uma mensagem do Windows será enviada ao formulário sempre que este evento de arrastar arquivos ocorrer. Deve-se criar uma procedure para interceptar essa mensagem.

Procedures que interpretam mensagens da API têm uma assinatura diferente e possuem a palavra reservada "message" no final. A mensagem em questão é a WM_DROPFILES e a assinatura da procedure fica como é mostrado a seguir:

```
procedure WMDropFiles(var Msg: TWMDropFiles); message WM_DROPFILES;
```

Já sua implementação consta na **Listagem 8**.

O método AdicionaIconeArrastado aceita como parâmetro o ícone temporário criado pelo WMDropFiles e tem como responsabilidade obter as informações desse ícone e decompô-las de forma a gravar no ClientDataset.

A **Listagem 9** mostra como funciona o método AdicionaIconeArrastado.

Ainda é necessário explicar os métodos estáticos públicos globais FileNameNoExt e InfoAtalho. FilenameNoExt, dado um caminho completo, pegará o nome de um arquivo sem o caminho e sem a extensão. É usado para a descrição do ícone caso ele não tenha uma descrição no Windows.

FilenameNoExt não funcionará com nomes que tenham mais de um ponto no meio do nome. Uma maneira de fazer isso é reescrever essa função contando do último ao primeiro caractere e parando no primeiro ponto, depois obtendo o restante. Esse é mais um refactoring a ser feito futuramente.

A procedure InfoAtalho aceita um caminho como primeiro parâmetro, que é o caminho do atalho.

Nota

No Windows atalhos também são arquivos, com extensão lnk. Esses arquivos possuem um caminho que aponta para outro arquivo (o destino do atalho), possuem um ícone e podem ou não possuir uma descrição. Por exemplo a descrição de um atalho para o Browser pode ser "acesse a internet".

A InfoAtalho tem dois parâmetros marcados como "out". Na prática "out" tem quase o mesmo efeito de "var" com uma diferença: os valores das variáveis são inicializados automaticamente na chamada da procedure, o que faz com que os valores iniciais sejam ignorados.

Use a palavra "out" sempre que uma variável for estritamente de saída e seu valor inicial obrigatoriamente for ignorado, e não trabalhe com esse valor inicial dentro da procedure. Isso é bom para a leitura do código, pois só de olhar para esses parâmetros

Listagem 8. Message WMDropFiles

```
procedure TfrmSideBar.WMDropFiles(var Msg: TWMDropFiles);
var
 i: Integer;
 NumArqs: Integer;
 NomeArq: PChar;
 StrArq: string;
 IconeTemporario: TIconeComando;
 tmpCaminho, tmpDescricao: string;
 pt: TPoint;
begin
 pt := Mouse.CursorPos;
 NumArqs := DragQueryFile(Msg.Drop, $FFFFFFFF, PChar(StrArq), 255);
 for i := 0 to NumArqs - 1 do
 begin
  //DEVE-SE LIMPAR E SETAR O VALOR MAXIMO DA STRING A CADA ITERAÇÃO.
  StrArq := '';
  SetLength(StrArq, MAX_PATH);
  DragQueryFile(Msg.Drop, i, PChar(StrArq), 255);
  IconeTemporario := TIconeComando.create(nil);
  with IconeTemporario do
  begin
   Top := pt.Y + (i * TAMANHOICONE); //este TAMANHOICONE é arbitrário e
   corresponde ao tamanho de um lado do ícone
   Left := ICONELEFT;
   //verificar se o arquivo é um atalho
   if (AnsiSameText('.lnk',ExtractFileExt(strArq))) then
   begin
    InfoAtalho(strArq, tmpCaminho, tmpDescricao);
    Comando := tmpCaminho;
    Descricao := tmpDescricao;
    if Descricao = '' then
     Descricao  := FileNameNoExt(StrArq);
   end
   else
   begin
    Comando := strArq;
    Descricao := FileNameNoExt(StrArq);
   end;
   Parent := Self;
   PopupMenu := pmIcones;
  end;
  AdicionaIconeArrastado(IconeTemporario);
  FlistaIcones.Add(IconeTemporario);
  FListaIcones.OrganizaIcones;
 end;
end;
```

Listagem 9. Método AdicionaIconeArrastado

```
procedure TfrmSideBar.AdicionaIconeArrastado(Icone: TIconeComando);
var
 msg: string;
begin
 try
  if not cdsIconesInicia.Active then
   cdsIconesInicia.Open;
  cdsIconesInicia.Insert;
  cdsIconesIniciaCOMANDO.AsString := icone.Comando;
  cdsIconesIniciaDESCRICAO.AsString := icone.Descricao;
  cdsIconesIniciaVERTICAL.AsInteger := icone.Top;
  cdsIconesInicia.Post;
 except
  on e: Exception do
  begin
   Icone.Free;
   msg := 'Ocorreu um erro durante a criação de um dos ícones.' +
    #13#10 + e.Message;
   MessageBox(0, PChar(msg), 'Erro ao criar ícone', MB_ICONEXCLAMATION);
  end;
 end;
end;
```

"out" dá para saber que essa procedure devolverá duas coisas para quem a chamar: o caminho para onde o atalho aponta e a descrição do atalho, se houver. A FilenameNoExt e a InfoAtalho podem ser vistas no código disponível para download.

A procedure InfoAtalho usa uma variável que implementa a interface IshellLink e outra que implementa IpersistFile. Sl é instanciado com CoCreateInstance(CLSID_ShellLink,nil,CLSCTX_INPROC_SERVER,IShellLink,sl). As variáveis cam e descry são arrays de caracteres e serão usadas no lugar de ponteiros para obter, via API do Windows, os valores procurados. Antes disso, porém, elas são inicializadas com FillChar, que preenche com #0 toda a cadeia. Isso faz com que ela seja terminada com #0 mesmo que nem toda a sua extensão seja usada.

O atalho é carregado com pf.Load como se fosse um persistentfile ou um arquivo ini. Com isso o arquivo de atalho pode ser lido. Outras alternativas para STGM_READ são:

```
{$EXTERNALSYM STGM_READ}
STGM_READ          = $00000000;
{$EXTERNALSYM STGM_WRITE}
STGM_WRITE         = $00000001;
{$EXTERNALSYM STGM_READWRITE}
STGM_READWRITE     = $00000002;
```

Para ler, escrever e ambos. Há ainda outras opções de escrita e leitura que podem ser vistas na unit ActiveX. Pode-se usar as permissões de escrita para mudar o destino de um atalho, ou criar um.

Nota

Tudo o que o método InfoAtalho faz usa ActiveX e Com para ler ícones de atalho do shell.

Depois disso os métodos GetDescription e GetPath do sl podem ser chamados. No caso do GetDescription apenas é passado o endereço do vetor @descr e o limite de caracteres, que é seu tamanho, 300. Passar um endereço para um vetor de caracteres é o mesmo que passar um pchar de tamanho fixo, mas sem a preocupação de alocar ou desalocar memória. O tamanho fixo pode ser uma vantagem ou desvantagem dependendo do contexto.

Já o método GetPath passa, além do ponteiro @cam e da constante MAX_PATH, wfs, que é uma variável da estrutura TWin32FindData usada na API do Windows para a busca ou pesquisa de arquivos e a constante SLGP_UNCPRIORITY, que tem o valor 2 e define como o caminho será representado. Outras alternativas para SLGP_UNCPRIORITY podem ser encontradas na unit ShlObj.

No final as variáveis cam e descr são convertidas para string e atribuídas aos parâmetros de saída. Um possível refactoring aqui é isolar os três métodos estáticos globais que tratam de API ou ActiveX em uma unit separada.

Aparência

Uma barra cinzenta e quadrada que cobre todo o lado direito do desktop não é uma deskbar ideal. Como este é um programa

de utilidade limitada e preço mais limitado ainda, sua aparência conta muito na hora de atrair a atenção de usuários. A aparência é uma parte importante da "usabilidade".

Até agora a barra peca no quesito aparência. Ela possui uma largura inconveniente, já que os ícones têm apenas 32 pixels de largura. No máximo uma margem de segurança de cada lado para os ícones ficarem centralizados estaria ótimo.

O programador pode definir se o formulário será do tipo fsStayOnTop ou não. Não sendo StayOnTop tem a vantagem de não cobrir a barra de ferramentas do Windows ou outros programas, mas tem a desvantagem de não estar sempre em foco e receber as mensagens OnMouseEnter e OnMouseLeave.

A barra totalmente quadrada e encostando nos limites da tela não é nada atraente. O ideal é dar alguns pixels de "respiro" de cada lado. Uma das maneiras de se deixar as bordas do formulário arredondadas é usar como imagem de fundo uma figura retangular com as bordas arredondadas. Essa figura deve ter uma cor padrão, convencionada, do lado de fora do retângulo. Essa cor deve ser a mesma cor do formulário e também a cor que será transparente, e a propriedade TransparentColor do formulário deve ser true.

A cor convencionada é $F600FF, que é um rosa tão berrante que dificilmente seria usado em algum outro lugar. Isso é apenas uma convenção, pode-se usar qualquer outra cor, mas é bom evitar usar o preto, pois com isso labels e outras coisas com texto ficariam com as letras transparentes.

Deve-se definir a propriedade color do formulário para $F600FF assim como a propriedade TransparentColorValue, ou seja, a propriedade color e a TransparentColorValue sendo que a mesma significa que todo o formulário será transparente e todo lugar na figura onde tiver a cor $F600FF também será. Também é necessário definir a propriedade TransparentColor do formulário para true.

Neste artigo foi usado como exemplo uma imagem com um retângulo contendo um degradê azul e cantos arredondados. O fundo da imagem é da cor "proibida". Este desenho foi feito no programa Paint.Net, e não tem nada de especial. Você pode pedir a um designer para fazer um desenho um pouco mais "profissional".

Para colocar essa imagem como fundo da tela deve-se usar o componente TjvBackground da biblioteca JEDI. A propriedade picture do componente deve carregar o arquivo do desenho feito, mostrado na figura. A propriedade TransparentColor do TJVBackground funciona diferente: o padrão dela não é RGB, é BGR. Isso significa que o código hexadecimal da cor deve ser invertido. Então se a cor transparente é $F600FF, a invertida é $FF00F6. Muita atenção nesse detalhe.

Existem outras formas de se fazer isso, olhando na internet é possível achar muitas formas de se definir uma imagem como background de um formulário sem usar a biblioteca JEDI, mas usando, por exemplo, o canvas do formulário. Seria interessante até se a Embarcadero já colocasse no TForm a propriedade BackgroundImage, assim não seriam necessários esses artifícios.

A **Figura 3** mostra o background com cantos arredondados e degradê que foi feito no paint.net. A figura foi encolhida no artigo, mas ela tem 1000 px de altura.

Figura 3. Background do formulário

Outra característica interessante será a propriedade Alphablend. Setada como true e definido um valor maior que zero e menor do que 255 é possível deixar o formulário semi-transparente.

O SpeedButton usado para a função de "travar" deve ser trazido mais para baixo e para a esquerda, para ficar dentro das bordas do desenho. Ele deve ser setado como transparente também para mostrar o fundo da form.

Um imagelist deve ser adicionado ao formulário com duas imagens: a imagem de uma "tachinha" solta e uma da "tachinha" pregada, para a função de travar/destravar. O botão agora, além de inverter o valor de FTravado, deve trocar o ícone do botão SpeedButton1 usando para isso as imagens gravadas no ImageList1. A imagem 0 é uma tachinha despregada e a imagem 1 é a tachinha pregada.

A imagem na **Figura 4** mostra a barra semitransparente azul, com cantos arredondados e o ícone da tachinha, além de alguns ícones adicionados.

Outra melhoria visual aqui cabe somente se o programador optou por organizar os ícones automaticamente ou se optou por ter apenas uma coluna de ícones: colocar uma constante na unidade dos ícones que será o "left" de todos eles, assim eles não ficam desalinhados verticalmente, mas ficam todos com o mesmo left.

Comportamento adicional

Embora atalhos para aplicações sejam a parte principal das "regras de negócio" de uma deskbar não é só isso que ela deve fazer. Ela deve iniciar junto com o Windows, e para isso deve-se adicionar uma entrada na chave "run" do registro ou no menu inicializar em iniciar → programas → inicializar.

Um instalador pode fazer isso durante a instalação, ou a própria deskbar durante a primeira execução. Para saber se é a primeira execução é fácil: a primeira vez que for executada não haverá o arquivo de dados.

Para se usar o registro deve-se incluir a unit registry no uses. Duas novas functions estáticas globais são adicionadas nesse ponto, uma para verificar se a entrada no registro já existe e outra para efetivamente registrar o programa. A **Listagem 10** mostra ambas.

Figura 4. Barra em funcionamento

Listagem 10. Funções para registrar o programa na inicialização

```
function Registrado(nomeChave: string): boolean;
var
 reg: TRegistry;
begin
 try
  Reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  result := true;
  if not reg.Openkey('\SOFTWARE\Microsoft\Windows\CurrentVersion
  \Run', true) then
  begin
   result := false;
   reg.Free;
   exit;
  end;
 finally
  Result := reg.ValueExists(nomechave);
  reg.Free;
 end;
end;


function RegistraInicializacao(nomeChave: string; nomeprog:
string = ''): Boolean;
var
 reg: TRegistry;
begin
 try
  Reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  if trim(nomeprog) = '' then
   nomeprog := paramstr(0); //application.exename;

  if not reg.Openkey('\SOFTWARE\Microsoft\Windows\CurrentVersion
  \Run', false) then
  begin
   result := false;
   reg.Free;
   exit;
  end;

  try
   if not Registrado(nomechave) then //verifica se existe o registro
   begin
    reg.WriteString(nomeChave, nomeprog);
    SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, 0);
    result := true;
   end;
  except
   result := false;
  end;
 finally
  reg.CloseKey;
  reg.Free;
 end;
end;
```

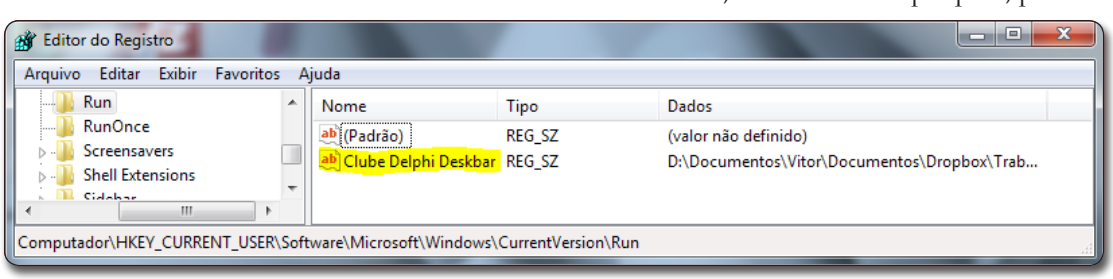


Figura 5. RegEdit

O método RegistraInicializacao usa o Registrado, por isso pode ser executado várias vezes com segurança. Ele aceita dois parâmetros, um é o nome que aparecerá no registro e o outro é o caminho do programa a ser executado. O segundo parâmetro pode ser omitido porque pode ser obtido automaticamente através de paramstr(0).

É possível ver no RegEdit e no Msconfig que uma entrada foi adicionada no registro, conforme as **Figuras 5** e **6**.

A chave usada foi HKEY_CURRENT_USER para que o programa seja registrado para cada usuário. O programa pode ter duas instâncias rodando caso existam dois usuários conectados na máquina mas um em "stand by", quando por exemplo ele clica em "trocar usuário" em vez de "logoff" para outro usar o computador.

Isso não interfere em nada pois cada um terá o seu arquivo de ícones. Uma pequena mudança é feita no método Inicializa, para executa o método RegistraInicializacao. Essa mudança está logo abaixo do segundo if e pode ser vista a seguir:

```
if not FileExists(NomeArquivo) then
begin
 RegistraInicializacao('Clube Delphi Deskbar');
 cdsIconesInicia.CreateDataSet;
end;
```

Não é interessante que certas coisas como inicialização junto com o Windows, sejam fixas, ainda falta uma janela de configuração para que isso possa ser opcional.

Um outro comportamento esperado é mostrar um campo para digitar um comando. Se for um comando que está salvo no cds então executa direto. Se for um comando que não está salvo, então abrir uma janela do browser padrão e fazer uma pesquisa. Um outro comportamento seria, antes de fazer a pesquisa, procurar

pelo disco rígido por um programa com nome semelhante e executá-lo, partindo para a busca apenas caso não encontre. Esse comportamento ficará para o próximo artigo, juntamente com buscas de arquivos.

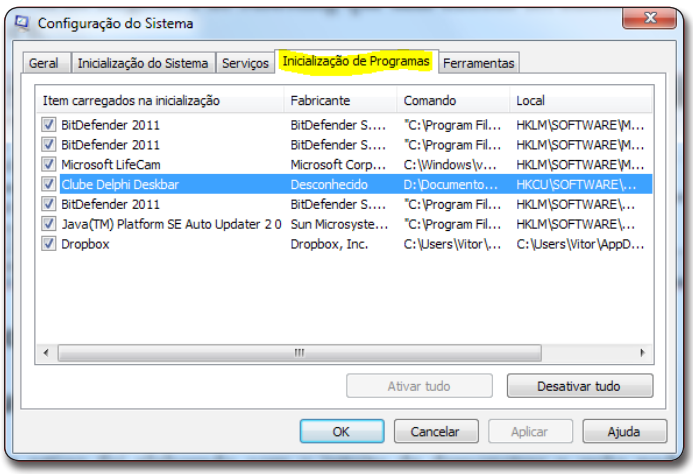


Figura 6. MsConfig

Para isso foi criado um formulário horizontal com um campo de texto onde será digitado o "comando" a ser executado. Também haverá uma cópia do componente ClientDataSet do formulário principal, que pode ser uma cópia ctrl-c ctrl-v mesmo. No evento OnKeyPress deste edit será feita uma busca pelo comando no cds. Ao se pressionar enter o comando encontrado deve ser executado. Se o comando não for encontrado então executa-se uma pesquisa com o browser padrão.

Esta janela não será criada junto com a aplicação, então deve ser removida do Auto-Create forms. Para a execução desse formulário deve haver uma hotkey, que pode inclusive ser configurável. Esta hotkey pode ser inserida no formulário através de um TActionList, com uma action chamada "Executar". Esta pode inclusive ser adicionada ao popupmenu ligado ao trayicon. A **Listagem 11** mostra o código desse formulário e a **Figura 7** apresenta o seu design.

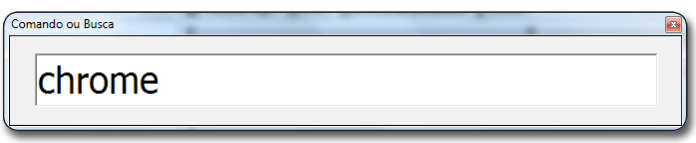


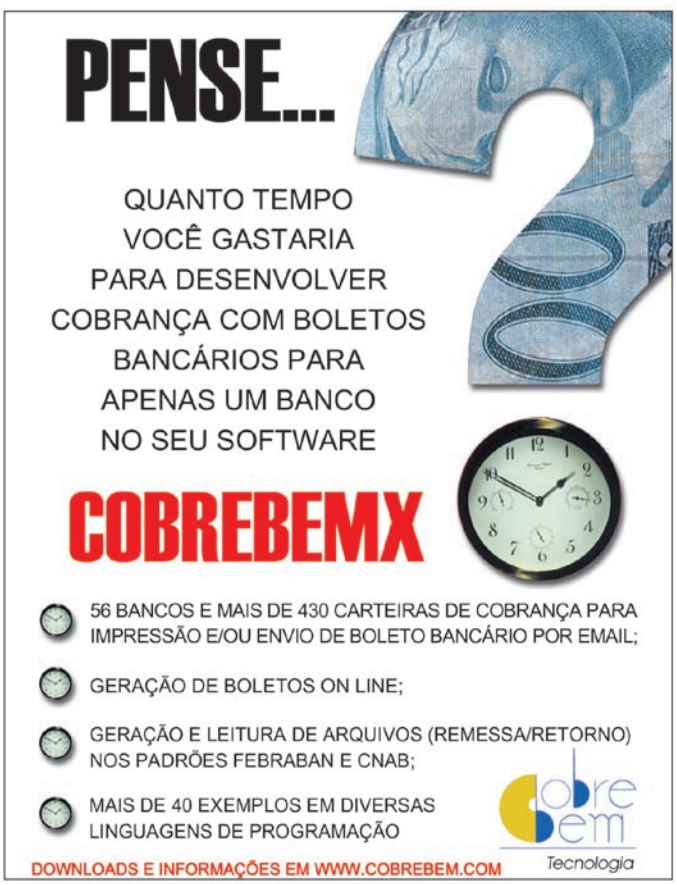
Figura 7. Janela de busca

Como exemplo foi feita uma pesquisa da palavra "chrome" e o browser chrome, que estava adicionado na barra lateral, abriu. Quando é feita uma pesquisa "cromo" então é aberto o browser padrão fazendo uma pesquisa sobre cromo no google. O provedor de busca está hard-coded no código fonte, mas pode ser feito de modo a ser modificado via configuração também.

O código da **Listagem 11** é muito simples, ele abre o arquivo .cds com os comandos armazenados e no keypress do edit, caso a tecla pressionada seja #13, que é o enter, ele procura pela palavra na lista de comandos usando para isso um filtro "like". Se ele encontrar ele executa o primeiro que encontrou, se não encontrar abre o browser padrão e faz uma busca.

Listagem 11. Barra de comando ou busca

```
procedure TfrmComandoOuBusca.edComandoKeyPress(Sender: TObject; var
Key: Char);
begin
 if key = #13 then
 begin
  cdsIconesInicia.Filtered := False;
  cdsIconesInicia.Filter := 'COMANDO like ''%'+edComando.Text+'%''';
  cdsIconesInicia.FilterOptions := [foCaseInsensitive];
  cdsIconesInicia.Filtered := true;
  if cdsIconesInicia.RecordCount > 0 then
   ShellExecute(0, nil, pchar(cdsIconesIniciaCOMANDO.AsString), nil, nil,
   SW_SHOWNORMAL)
  else
   ShellExecute(Application.Handle,'open',
    PChar('http://www.google.com.br/search?q='+edComando.Text),
    nil, nil, SW_SHOWNORMAL);
 end;
end;
procedure TfrmComandoOuBusca.FormCreate(Sender: TObject);
var
 NomeArquivo: string;
 DiretorioArquivo: string;
begin
 DiretorioArquivo := ExtractFilePath( ParamStr(0) )+'\'+ NomeUsuario+'\';
 NomeArquivo :=  DiretorioArquivo + 'icones.cds';
 cdsIconesInicia.FileName := NomeArquivo;
 cdsIconesInicia.Open;
end;
```

Refactoring

Refactoring ou refatoração é o ato de propositalmente refazer um código que está funcionando para torná-lo mais legível e elegante, ou mais performático quando necessário.

Note que muitas vezes para tornar um código mais performático é necessário usar truques e conhecimentos profundos que podem ser obscuros para a maioria das pessoas. Esse tipo de código deve obrigatoriamente ser muito bem documentado e explicado, e é recomendável apenas onde há reais problemas de performance e a mesma é estritamente necessária.

Na grande maioria dos outros casos uma refatoração é feita para assegurar a legibilidade do código, torná-lo mais legível e elegante para seres humanos e, quem sabe, encontrar e exterminar bugs, leaks e outras ameaças.

Refactoring sempre é uma boa prática, adotada principalmente em metodologias ágeis como SCRUM ou XP como uma religião e respeitada por todas as outras metodologias (ou na falta delas). A grande questão é: se você faz refactoring, ponto positivo.

Existe um conjunto de operações básicas, atômicas, no código que em conjunto formam os refactorings. Dentre estas destacam-se: extrair método, extrair interface, extrair classe e mover método. A lista detalhada de operações e o que elas fazem podem ser encontradas em livros sobre refatoração. O Delphi possui um menu de contexto chamado refactoring que automatiza muitas dessas operações.

Assim como existem design patterns, que são padrões de projetos, existem "antipatterns" que são chamados bad smells, ou "fedores" no código.

Ao longo desse artigo muito código "fedorento" foi propositalmente colocado na deskbar para que terminasse mais rápido com as funcionalidades desejadas. A filosofia seguida foi: faça funcionar primeiro, faça bonito depois, faça rápido por último.

Nota

Padrões de projeto são uma tradução de design patterns e a palavra padrão aqui não tem o significado de norma, como lei, mas de padrão no sentido de repetição em um desenho, como um tecido que pode ser listrado ou quadriculado. É esse o sentido de padrões de projeto: um "desenho" que é possível de ver "repetido" ao longo de um ou vários projetos.

Não que essa filosofia seja correta, é apenas mais uma maneira de fazer ou de não fazer um projeto, dependendo muito do projeto. Uma deskbar/sidebar para o desktop não é bem um monitor de caldeira atômica ou um sistema de contas de investimento.

Alguns bad smells são bastante conhecidos, em português, e na **Tabela 1** há a lista de alguns e onde foram introduzidos.

Mais uma vez, uma lista completa de "maus cheiros" pode ser encontrada em livros e artigos sobre refactoring. Reconhecer e saber operar em todos esses "fedores" é uma arte que leva tempo para aprimorar, mas que vale a pena pela qualidade e manutenibilidade resultante.

Programadores iniciantes, principalmente em ferramentas RAD como o Delphi, têm a tendência de criar esses maus cheiros por copiar e colar código, fazer uma unit de um formulário usar outra apenas para não ter que passar mais parâmetros em métodos, criar acoplamentos desnecessários e assim por diante. Muito provavelmente você já fez melhorias em códigos desse tipo sem se dar conta de que estava fazendo um refactoring.

Dois refactorings fáceis que podem ser feitos rapidamente com o menu refactor do Delphi nesse momento é extrair a chave do registro para a inicialização do deskbar movendo-a para um resourcestring. Isso pode ser feito com a url do buscador também, se desejado, mas no próximo artigo isso será configurável.

Bad Smell	O que é	Onde foi inserido
Código Duplicado	Código idêntico ou similar em dois ou mais lugares ao mesmo tempo.	O mesmo cds copiado em dois formulários, e a lógica para execução de um comando reproduzida no formulário de comando/busca.
Método muito longo	Método macarrônico que parece fazer tudo	WMDropFiles
Classe muito grande	Classe muito grande que parece conter responsabilidades que poderiam ser divididas com outras classes	Em certa medida a classe ícone, mas principalmente o formulário principal, que pode ser dividido em duas ou mais classes.
Classe preguiçosa	Classe que parece fazer muito menos do que poderia ou deveria	A classe TIconeList poderia ser mais robusta, como ela mesma controlar o arquivo .cds
Identificadores muito longos	Convenções de codificação podem ajudar a evitar nomes muito longos sem que percam o sentido	-
Uso excessivo de literais	Números e strings literais podem ser colocados em constantes para melhorar a legibilidade do código ou então colocados em arquivos externos para ser encontrados.	Endereço do buscador para abrir no browser.
Intimidade inapropriada	Quando uma classe depende muito de detalhes de implementação de outra.	A TIconeComando depende muito de como a TIconeList os organiza.
Inveja de Dados	Quando uma classe usa muito propriedades de uma outra, e precisa ler muitas propriedades antes de executar uma ação.	O formulário principal usa muito a TIconeComando, mas esse é um bad smell também presente no restante do código.
Comentários	Comentários são importantes, mas eles não devem ser demasiados ou óbvios para não "sujar" o código. Tampouco devem trechos de código serem comentados apenas para serem desativados temporariamente e reusados quando necessário.	Alguns trechos de código foram comentados apenas para não serem executados, o que leva a necessidade de uma recompilação quando esse código voltar a ser necessário.

Tabela 1. Identificação dos códigos ruins

Para isso selecione a string \SOFTWARE\Microsoft\Windows\CurrentVersion\Run e use a opção Refactor>extract resource string. Use no outro método que tem a mesma chave de registro o mesmo nome que foi dado a essa resourcestring. A outra refatoração consiste em selecionar dentro de WMDropFiles o trecho da **Listagem 12** e usar a opção extract method.

Isso se transformará no método CriaIcone, e o WMDropFiles será modificado conforme código a seguir:

```
DragQueryFile(Msg.Drop, i, PChar(StrArq), 255);
CriaIcone(IconeTemporario, StrArq, i, pt);
AdicionaIconeArrastado(IconeTemporario);
FListaIcones.Add(IconeTemporario);
FListaIcones.OrganizaIcones;
```

Porém isso pode ficar ainda melhor: em vez de o ícone ser passado como parâmetro para CriaIcone e o mesmo criá-lo internamente, esse parâmetro pode ser omitido e o método CriaIcone se transformar em uma function que retorna um ícone. A **Figura 8** mostra a janela Extract Method.

Listagem 12. Trecho a ser extraido

```
IconeTemporario := TIconeComando.create(nil);
with IconeTemporario do
begin
  Top := pt.Y + (i * TAMANHOICONE); //este TAMANHOICONE é arbitrário e
  corresponde ao tamanho de um lado do ícone
  Left := ICONELEFT;
  //verificar se o arquivo é um atalho
  if (AnsiSameText('.lnk',ExtractFileExt(strArq))) then
  begin
    InfoAtalho(strArq, tmpCaminho, tmpDescricao);
    Comando := tmpCaminho;
    Descricao := tmpDescricao;
    if Descricao = '' then
      Descricao := FileNameNoExt(StrArq);
  end
  else
  begin
    Comando := strArq;
    Descricao := FileNameNoExt(StrArq);
  end;
  Parent := Self;
  PopupMenu := pmIcones;
end;
```

Conclusão

Este artigo foi elaborado com o intuito de demonstrar como pode ser difícil uma aplicação que tem uma funcionalidade aparentemente trivial. Ele introduz o leitor em POO mostrando conceitos de agregação, composição, dependência, herança e polimorfismo e também mostra algumas questões de usabilidade e design.

Após a leitura deste artigo o leitor poderá criar aplicações com seus próprios ícones e disparar comandos através deles. Isso pode incrementar a usabilidade do seu sistema, e incrementando a usabilidade você supera as expectativas dos usuário, fornecendo assim um ponto positivo a favor do seu sistema quando o usuário está para decidir entre vários.

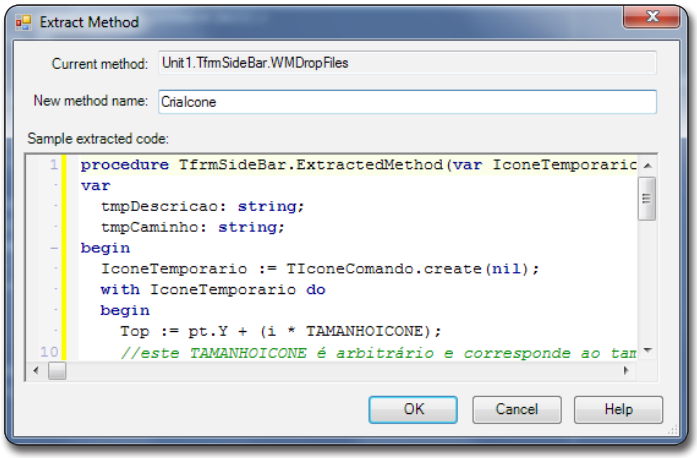


Figura 8. Extract Method

Entretanto o ponto chave deste artigo é a introdução à refatoração. Há vários pontos neste código que não estão errados em si, mas podem ser melhorados para assegurar a manutenibilidade e extensibilidade do código.

No próximo artigo será mostrado como separar o controle dos ícones do formulário da deskbar, tornando possível adicionar ícones a painéis ou forms em outros projetos. Também algumas units serão separadas em duas para uma melhor organização do código. Também será mostrado como usar esses ícones em uma form principal de uma aplicação grande, usando os ícones tanto para executar tarefas externas do Windows como abrir forms da própria aplicação, usando para isso um pouco de metaprogramação.

Vitor Luiz Rubio
vitorrubio@gmail.com - http://blog.vitorrubio.com.br
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Blogs do autor
http://blog.vitorrubio.com.br
http://twitter.com/vitorrubio

Paint .Net
http://paint.net/

Projeto JEDI
http://www.delphi-jedi.org/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Os padrões de arquitet

Resumo DevMan

De que trata o artigo:

Este artigo fará um apanhado geral dos três padrões de arquitetura mais utilizados no desenvolvimento de software: MVC, MVP e MVVM. Serão vistos o significado de cada uma destas siglas e que tipo de problemas estes padrões visam resolver. Na medida em que forem apresentados, será elucidado em que tipo de tecnologia cada um melhor se encaixa. De antemão, pode-se citar como exemplo o padrão MVVM, que melhor se enquadra em aplicações WPF/Silverlight.

Em que situação o tema é útil:

O que as siglas MVC, MVP e MVVM têm em comum é, além do fato de cada uma representar um padrão de arquitetura, todas têm como objetivo central isolar ao máximo a camada de apresentação de um sistema. Além disso, em termos de nomenclatura, todas coincidem em suas duas letras iniciais: M e V, que se referem às camadas Model (representação do domínio do sistema) e View (representação gráfica do modelo), respectivamente. Dessa forma, presume-se que o que varia entre estes padrões é o conceito em torno da outra letra restante. Todas estas constatações serão melhor detalhadas conforme o decorrer do artigo.

Os Padrões de Arquitetura MVC, MVP e MVVM:

Em desenvolvimento de software, um padrão de arquitetura, a grosso modo, faz referência à forma de construção de um sistema. Para os desenvolvedores Delphi, principalmente depois do lançamento do Delphi Prism, que apresentou novas possibilidades devido à integração com o .NET Framework, três deste padrões já se tornaram uma constante em seu processo de desenvolvimento. São eles: MVC, MVP e MVVM, acrônimos de Model-View-Controller, Model-View-Presenter e Model-View-ViewModel, respectivamente. Em vista disso, o presente artigo visa apresentar uma explanação sobre cada um deles, com a intenção maior de atingir principalmente o público que está iniciando no desenvolvimento, seja com o Delphi tradicional ou Delphi Prism (em vista dos recentes artigos que esta revista tem apresentado sobre o assunto). Para este último, a abordagem se dará de forma mais teórica, já que em termos práticos, o uso destes padrões com o .NET Framework envolve, na maioria dos casos, outras tecnologias, frameworks e conceitos, que exigiriam artigos específicos sobre cada um. Dessa forma, a parte prática do artigo demonstrará a utilização de um destes padrões em um projeto Delphi Win32 tradicional. Esta escolha é justificada pelo fato de que para esse tipo de aplicação não é muito comum o uso destes padrões, devido a uma série de motivos. Dentre estes, pode-se citar a escassez de material sobre o assunto, que é o que o presente artigo visa sanar.

A construção de um software expõe aos desenvolvedores duas vertentes internas distintas, no que diz respeito às suas responsabilidades perante o produto final. Estas duas vertentes, ou partes, são conhecidas como modelo e visão. Um modelo é a representação do software mediante a resolução de um determinado problema, ou seja, é a visualização do sistema que se está construindo a fim de proporcionar um melhor entendimento. Já a visão é a porta de entrada para que o usuário possa interagir com este modelo, sem a necessidade de conhecimento de codificação para utilizá-lo. Outra função importante realizada por ela é a de representar visualmente os dados do modelo para o usuário. Conceitualmente falando, o ideal é manter estas duas partes do sistema desconectadas, como se fossem subsistemas. Dessa forma, uma visão poderia ser facilmente substituída a qualquer tempo, enquanto que um modelo bem definido poderia ser reutilizado em distintos

ura MVC, MVP e MVVM

Fabrício Hissao Kawata

projetos. Além disso, para que esta separação ocorra e se estabeleça, espera-se que não seja necessário um alto índice de trabalho, pois um processo muito trabalhoso poderia desencorajar a sua utilização.

Mesmo com os papéis bem definidos e dada a importância em manter as partes separadas, é comum encontrar situações em que o conceito é violado. Dessa forma, começa haver uma mescla entre as partes, ocasionando dependências desnecessárias e trazendo problemas futuros, imperceptíveis num primeiro momento, como prejudicar a manutenção de código e até mesmo a qualidade do produto final. Isto ocorre devido a fatores variados, sendo o principal deles a pressão por parte dos envolvidos (gerente e cliente), em vista do cumprimento de prazos.

SoC

SoC é o acrônimo de Separation of Concerns que, em português, poderia ser livremente traduzido como Separação de Interesses. Este termo, apesar de muitos não conhecerem, não se trata de um novo termo no jargão da ciência da computação. SoC já é um termo recorrente, cujo significado é simples: garantir que o código tenha um único e bem definido propósito (objetivo). Isso significa que o mesmo deve seguir uma linha de responsabilidades, evitando desvios e bifurcações. Esta separação de interesses é aplicável em todos os níveis de código, desde um simples método do sistema até as partes que o compõem.

O principal objetivo do SoC é limitar dependências e, nos casos em que seja imprescindível a existência de uma, esta dependência deve ser abstraída ao máximo. Isto se justifica pelo fato de que códigos demasiadamente interdependentes são mais difíceis de manter. Uma simples alteração pode significar a quebra de outras partes do sistema, em decorrência de uma possível dependência. Um exemplo mais grave é a existência de uma dependência de caráter cíclico, onde duas partes do sistema (duas classes, por exemplo) são mutuamente dependentes entre si, impedindo um possível crescimento natural de uma parte sem que este processo afete a outra parte.

Este tipo de problema pode ser resolvido ao se garantir que as dependências sejam dirigidas a uma hierarquia vertical de baixo para cima onde, por exemplo, um código de nível mais elevado estabeleça uma dependência com código de níveis inferiores.

Toda esta teoria em busca de uma melhor forma de desenvolver software é fruto de estudos realizados ao longo dos anos que culminaram, entre outras coisas, na criação dos chamados Padrões de Projeto. Cada padrão deste, basicamente, visa solução de um problema recorrente, que é o que acontece com os padrões de arquitetura MVC, MVP e MVVM, como serão vistos adiante.

Model-View-X

Tanto MVC, MVP quanto MVVM seguem a convenção Model-View-*, onde * é a camada mediadora entre modelo e visão. Isto se deve ao fato de que, uma vez estabelecido que Model e View são as duas camadas sistêmicas principais (podendo serem interpretadas inclusive como subsistemas), as variações destes padrões se dão por conta da camada intermediária. Basicamente, esta camada mediadora se difere na nomenclatura e em alguns detalhes de sua implementação, dentre as opções disponíveis para estes tipos de padrões. Todavia, sua responsabilidade principal permanece inalterada em todas as suas variações, que é a de receber as entradas de solicitações do usuário e consequentemente fornecer os dados que a camada de visão necessite. A **Figura 1** mostra, de forma abrangente, a variação da camada intermediária existente entre os padrões MVC, MVP e MVVM.

Ainda sobre a estrutura destes padrões, toda a justificativa em se adotar este nível de separação entre as partes do sistema gira em torno de duas palavras: acoplamento e coesão.

Em termos de desenvolvimento de software, acoplamento é a métrica utilizada para se mensurar a intensidade de dependência entre os módulos do sistema. Nestes casos, o que se espera atingir, é a definição de módulos "fracamente acoplados", em detrimento aos "fortemente acoplados". Diz-se que dois módulos estão "fortemente acoplados", quando há um alto grau de dependência entre eles, de tal forma que uma mudança em um dos módulos torne necessária a intervenção no outro. Neste caso, o ideal é que eles sejam "fracamente acoplados", para que as mudanças ocasionadas em cada um sejam isoladas, sem causar um impacto concomitante.

Já a coesão mede a consistência destes módulos, de forma individual. Um módulo "altamente coeso" é o ideal, e um "fracamente coeso" é o que se espera evitar. Diz-se que um módulo é fracamente coeso, ou seja, possui uma fraca coesão, quando realiza mais funções do que deveria. Para se estabelecer quais funções devem ou não ser realizadas pelo módulo basta ter em mente que um módulo "altamente coeso" é aquele que possui e se concentra em apenas um único objetivo principal.

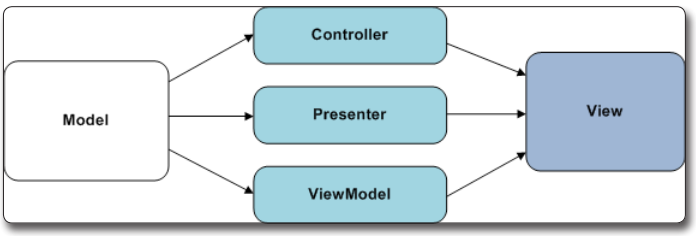


Figura 1. Padrões MVC, MVP e MVVM

MVC

O padrão MVC ou Model-View-Controller, popularmente falando, pode ser considerado o ancestral de todos os demais padrões que serão abordados nesse artigo (MVP e MVVM). Seu surgimento se deu ainda na década de 80, se tornando um padrão muito popular para a criação de aplicações Web. A **Figura 2** mostra um esboço gráfico do MVC, bem como o relacionamento entre suas partes. Em termos práticos, basicamente, no MVC o Controller é definido como o ponto de entrada para as requisições de usuário, decidindo inclusive como lidar com cada entrada deste. Este mesmo Controller acessa então a camada Model, que tem a incumbência de processar a requisição, cujo resultado final é exibido na camada View.

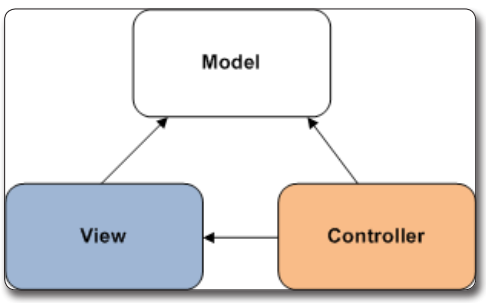


Figura 2. Padrão MVC

No cenário do Delphi, o MVC ganhou uma notoriedade recente com o Delphi Prism, visto seu leque de novas possibilidades, devido a integração com o .NET Framework. Isso se explica pelo surgimento do ASP.NET MVC, que é o framework da Microsoft voltado para a construção de aplicações Web baseadas no padrão MVC.

MVP

No MVP (ou Model-View-Presenter), a camada intermediária entre Model e View é denominada Presenter. Já fazendo um comparativo com o padrão MVC, no MVP tem a View como ponto de entrada para as solicitações do usuário, para então repassá-las à camada Presenter, que irá então repassá-las à Model. Sendo assim, Presenter deve ter o conhecimento das duas extremidades (View e Model). Uma característica peculiar do padrão MVP é que a camada Presenter não está associada diretamente à implementação da camada View, ao invés disso, ela mantém uma referência para uma interface que a represente. Por sua vez, opcionalmente, a View fará referência à Presenter também por interface (IPresenter).

Na **Figura 3** está a representação gráfica do esquema do padrão MVP e suas referências citadas.

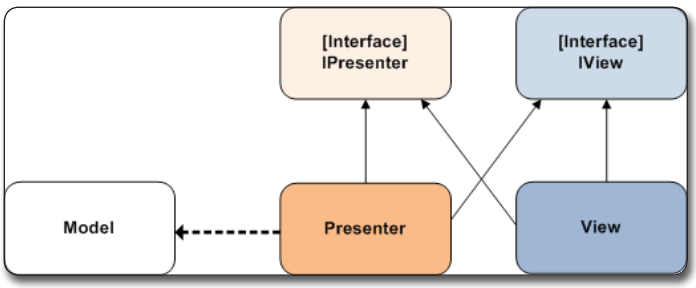


Figura 3. Padrão MVP

Nota

System.Web.UI.Page e System.Windows.Forms.Form são classes pertencentes ao .NET Framework. A primeira representa os arquivos .aspx, que nada mais são que as página ASP.NET Web Forms. Já a segunda representa a janela (Form) que constitui a interface de usuário de uma aplicação Windows Forms.

De acordo com o que foi visto, pode-se facilmente supor que em um sistema que faz uso do MVP teremos a presença de módulos extras (IView e IPresenter) para cada implementação de View e Presenter, respectivamente. Desta forma, tradicionalmente, poderemos ter uma estrutura em que para cada interface de usuário será necessária a construção destes quatro módulos. Isto, inevitavelmente, irá refletir em uma carga de trabalho maior, em termos de codificação.

Em termos de usabilidade, a arquitetura descrita em MVP pode ser utilizada em projetos Delphi Win32, porém é mais amplamente utilizada em projetos relacionados ao .NET Framework, como ASP.NET e Windows Forms. Por exemplo, para o primeiro caso, poderia se implementar para cada interface IView uma classe que herde de System.Web.UI.Page (para Web) ou System.Windows.Forms.Form (para Desktop), da mesma forma que TForm para projetos Delphi.

MVVM

MVVM, acrônimo para Model-View-ViewModel, é o caçula dos padrões apresentados, e é baseado em outro padrão, denominado Presentation Model, criado por Martin Fowler. Ele foi desenvolvido por John Gossman, então arquiteto WPF (Windows Presentation Foundation) da Microsoft, no ano de 2005. Sua concepção pode ser justificada pela intenção de aproveitar os novos recursos proporcionados pelo WPF, como binding e comandas (os quais já foram abordados em outros artigos nesta revista), que poderiam também ser utilizados para a separação de responsabilidades de um sistema. Só para citar, em função deles, a comunicação entre View e ViewModel pode ser configurada no próprio código XAML da aplicação. Um resumo de sua estrutura pode ser visto na **Figura 4**.

Conforme pode ser visto, neste padrão, a camada de visão, ou seja, a View, não possui qualquer conhecimento sobre a camada de modelo (Model) da aplicação. E esta última não possui dependência

com qualquer outra camada, ficando totalmente isolada. Toda a interação que a View necessite estabelecer com a Model é provida pela camada intermediária denominada ViewModel que, aliás, é peça chave neste padrão.

Informalmente falando, quando a camada de visão necessita de algo, este "pedido" é feito à ViewModel, que tem a incumbência de adquirir junto ao modelo tudo o que é necessário (por exemplo dados) para se atender a solicitação. A seguir a ViewModel se encarrega de enviar uma resposta à View, de modo que esta possa compreender e utilizar o que lhe foi repassado. Um ponto importante a ser ressaltado é que a camada ViewModel não deve ser em hipótese alguma ignorada, mesmo que a tentação em se acessar as classes do modelo diretamente pela camada de visão seja grande.

O padrão MVP na prática

Como dito no início do artigo, a parte prática abordará essencialmente o emprego de um dos padrões em um projeto Win32, utilizando o próprio Delphi. A escolha por este tipo de abordagem se deve ao fato de que muito desenvolvedor Delphi tem dúvidas sobre a real possibilidade da utilização destes padrões de arquitetura em uma aplicação tradicional Desktop. Isto pode ser comprovado nos fóruns sobre a tecnologia e afins. Assim sendo, o padrão a ser utilizado neste exemplo será o MVP, através de uma aplicação simples e didática, que visa fundamentalmente demonstrar a divisão das camadas estabelecidas pelo padrão.

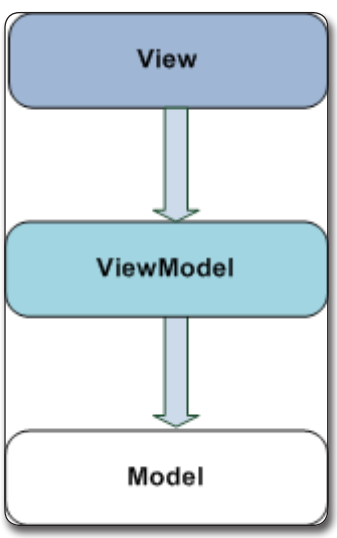


Figura 4. Padrão MVVM

Outra observação a ser feita é sobre a escolha do padrão MVP em detrimento ao MVC e MVVM. Como mencionado, estes três padrões já estão muito bem estabelecidos no desenvolvimento Delphi / Delphi Prism. No entanto, no cenário atual, MVC e MVVM estão mais bem situados em contextos específicos, sendo muito trivial um exemplo prático genérico sobre ambos. Melhor explicando, hoje no desenvolvimento Delphi, quando se pensa em MVC, logo vem à tona o framework ASP.NET MVC, que já vem totalmente preparado e dedicado ao melhor emprego do padrão. Além do que, em artigos passados o emprego do MVC em projetos Delphi Win32 também já foi abordado. Da mesma forma, o MVVM, que é totalmente associado a projetos Silverlight e WPF.

Em vista do que foi apresentado sobre cada um até agora, será muito mais proveitoso discorrer sobre cada um em artigos futuros específicos, abordando assim todos os seus detalhes técnicos e conduzindo o leitor a ter um conhecimento mais abrangente sobre cada tema.

Assim sendo, a aplicação de exemplo, conforme dito no início será totalmente didática, a fim de mostrar o real funcionamento desta divisão do contexto em camadas. O emprego do padrão Model-View-Presenter se dará na construção de uma aplicação Win32 (Desktop) simples, que soma dois números e retorna o resultado da operação. Em termos visuais, o resultado final esperado é semelhante ao da **Figura 5**.

Se estivéssemos utilizando a abordagem tradicional, sem a utilização de padrões, apenas seria necessário montar o visual, codificar o evento do botão Somar e tudo estaria pronto. No entanto, o que queremos é ir um pouco além e fazer uso de boas práticas de desenvolvimento como, por exemplo, não misturar regras de negócio (no caso a soma em si) na camada visual (o formulário).

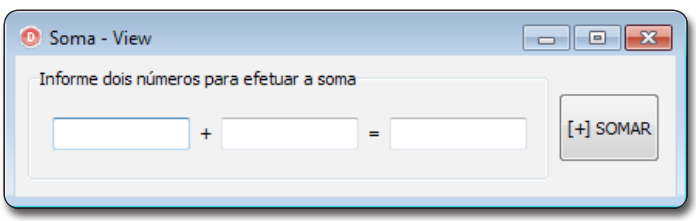


Figura 5. Visual da aplicação de exemplo

Nota do DevMan

O uso de Property (ou propriedades) no Delphi caracteriza uma boa prática de desenvolvimento, relacionado principalmente ao conceito da Orientação a Objetos. Através de uma property define-se um acesso controlado aos campos (fields) de uma classe, encapsulando-os ao meio externo. Dessa forma, se obtém um total controle de como os dados da classe serão lidos e escritos. A sintaxe básica para a definição de uma property pode ser vista na definição da classe deste exemplo, conforme a seguir:

```
property Valor1: Integer read FValor1 write SetValor1;
```

É importante ressaltar a presença das palavras-chave read e write, que indicam à property a origem dos valores de leitura e escrita, respectivamente. Neste caso em específico, ao se requisitar o valor da property Valor1, esta irá buscar o valor em FValor1. Já na atribuição de valor à propriedade, esta utilizará o método SetValor1.

Na classe Soma utilizada neste exemplo, a manipulação de seus campos internos privados FValor1 e FValor2 são providas pelas propriedades Valor1 e Valor2, públicas. Assim sendo, se num futuro for interessante estabelecer critérios específicos para a atribuição de valores para FValor1 e FValor2, isto poderá ser facilmente implantado nos métodos SetValor1 e SetValor2, que são os métodos de escrita utilizados pelas propriedades.

Model

Neste momento o que menos importa é o visual da aplicação. Portanto, com base na estrutura do padrão MVP apresentado na **Figura 3** o primeiro passo é a definição da camada Model, que representará as classes de negócio do sistema. É importante frisar que a todo o momento, durante o desenrolar desta parte prática, a estrutura descrita nesta figura será utilizada como guia para o desenvolvimento da aplicação de exemplo.

Já neste primeiro momento temos um grande diferencial se comparado a uma abordagem tradicional, pois estaremos definindo uma classe em vista da solução do problema e não mais tratando somente eventos de controles. A classe em questão é chamada de Soma e irá prover sua utilização pelas outras partes do sistema através de seus membros públicos, que estão representados por duas propriedades (property) e um método. As duas propriedades, do tipo inteiro, representam os argumentos a serem utilizados para a realização da operação de soma. Esta operação é definida como um método da classe, intuitivamente nomeado como Somar. A **Listagem 1** mostra um exemplo de como poderia ser feita a declaração desta classe, de acordo com a proposta apresentada.

Listagem 1. Classe Soma

```
TSoma = class
 private
  FValor1: Integer;
  FValor2: Integer;
  procedure SetValor1(const Value: Integer);
  procedure SetValor2(const Value: Integer);
  { private declarations }
 public
  { public declarations }
  property Valor1: Integer read FValor1 write SetValor1;
  property Valor2: Integer read FValor2 write SetValor2;
  function Somar: Integer;
 end;
 ....
procedure TSoma.SetValor1(const Value: Integer);
begin
 FValor1 := Value;
end;

procedure TSoma.SetValor2(const Value: Integer);
begin
 FValor2 := Value;
end;

function TSoma.Somar: Integer;
begin
 Result := FValor1 + FValor2;
end;
```

Interfaces – IPresenter e IView

Em programação, uma interface é a especificação de comportamentos abstratos que qualquer classe poderá implementar. Em outras palavras, ela rege as características de determinada classe, funcionando como um tipo de contrato, onde são especificados os atributos e métodos que obrigatoriamente deverão ser implementados pela classe. Isto porque, numa interface não há a implementação de qualquer método, somente a assinatura dos mesmos.

Conforme mencionado anteriormente, o padrão MVP faz uso de interfaces, de forma a isolar os detalhes da implementação de suas camadas Presenter e View. Baseado na estrutura do padrão, apresentado na **Figura 3**, vemos que o módulo Presenter é quem se relaciona diretamente com a Model. Porém, ainda conforme a estrutura apresentada, a implementação concreta de Presenter depende de uma Interface (IPresenter), que deverá ser codificada primeiramente. De forma semelhante se dá a construção da camada View, em que se faz necessária a presença também de uma Interface (IView). A codificação destas duas estruturas serão a base para as implementações concretas das classes Presenter e View do sistema.

Na **Listagem 2**, é apresentada a declaração da interface para a camada Presenter deste exemplo, nomeada como ISomaPresenter. Nesta estrutura é definida a especificação que a futura classe Presenter deverá possuir, como o método Somar (linha 9) que, como o próprio nome sugere, será usado para chamar o método Somar da classe TSoma do modelo.

Listagem 2. Interface ISomaPresenter

```
01 ISomaPresenter = interface
02   ['{3BED5DAB-58BE-458E-96F5-6F2299588E95}']
03   function GetModel: TSoma;
04   procedure SetModel(const Value: TSoma);
05
06   function GetView: ISomaView;
07   procedure SetView(const Value: ISomaView);
08
09   function Somar: Integer;
10   property Model: TSoma read GetModel write SetModel;
11   property View: ISomaView read GetView write SetView;
12 end;
```

Listagem 3. Interface IView

```
01 ISomaView = interface
02   ['{1D8E2B35-6BB6-405A-A222-EFC090BEE4D1}']
03   function GetValor1: Integer;
04   procedure SetValor1(const Value: Integer);
05
06   function GetValor2: Integer;
07   procedure SetValor2(const Value: Integer);
08
09   function GetPresenter: ISomaPresenter;
10   procedure SetPresenter(const Value: ISomaPresenter);
11
12   property Valor1: Integer read GetValor1 write SetValor1;
13   property Valor2: Integer read GetValor2 write SetValor2;
14   property Presenter: ISomaPresenter read GetPresenter
15    write SetPresenter;
16
17   function MostrarView: TModalResult;
18 end;
```

Apesar de simples, cabe fazer algumas observações com relação ao código apresentado. Como vimos, Presenter deverá ter conhecimento tanto de Model quanto de View, estabelecendo

um elo entre elas. Portanto, nesta Interface foram definidas duas propertys, Model e View, que serão usadas para referenciar ambas as camadas (linhas 10 e 11). A property Model é do tipo da classe codificada anteriormente, TSoma. Já a property View é do tipo da Interface utilizada pela camada de visão, ISomaView, que será construída a seguir.

Repare que fazendo isso estamos fazendo a interface não depender de uma implementação concreta como, por exemplo, uma classe TSomaView. Ao invés disso, ela faz referência a outra interface e assim, na codificação do sistema poderá se referenciar qualquer classe que a implemente. Isto também poderia ser feito com a property Model, tornando-a do tipo ISoma e fazendo a classe TSoma implementar esta interface. No entanto, esse detalhe não é vital ao emprego do padrão e no final das contas só agregaria uma complexidade a mais ao exemplo. Assim sendo, iremos seguir à risca a estrutura apresentada na **Figura 3**.

Conforme visto, a camada Presenter não se comunica diretamente com a View, mas sim com a interface que ela implementa, que na imagem aparece como IView. Dessa forma, a interface para a camada de visão deste exemplo poderia ser construída conforme apresentado na **Listagem 3**.

A interface ISomaView apresenta a característica base que a futura classe TSomaView deverá ter. Como quesitos principais foram definidas três propriedades (property): Valor1, Valor2 e Presenter. As duas primeiras, conforme pode se presumir, armazenarão os valores informados pelo usuário através da interface gráfica do sistema. Já a property Presenter faz referência à camada Presenter, mais especificamente à interface anteriormente definida. Novamente estabelecemos que a estrutura não dependa de uma implementação concreta, e sim de uma interface.

Em termos de métodos, foram definidos um total de sete. No entanto, a maioria é referente aos métodos utilizados para a leitura e escrita de cada property, com exceção de MostrarView, que é um método de apoio que define a chamada da View em tela, conforme será visto adiante.

Nota do DevMan

Em interfaces, GUID é um identificador global único, de valor binário de 16 bytes, que é especificado no momento de sua declaração. Ele é representado por uma sequência de caracteres literais dispostos entre colchetes, conforme a seguir:

['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'].

Na IDE, a geração de um GUID se dá por meio das teclas Ctrl+Shift+G.

Presenter

Uma vez definida a interface de Presenter, se faz necessária a codificação da classe que irá implementá-la e que fará efetivamente o "meio-de-campo" entre o Model e a View. Com base na nomenclatura que está sendo usada, a classe será definida como TSomaPresenter. A **Listagem 4** mostra a declaração desta classe.

Conforme já foi dito, uma interface estabelece um contrato que a classe que a implementa deverá seguir, tanto que na estrutura da interface somente estão declarados as assinaturas de métodos e alguns atributos. Dessa forma, por TSomaPresenter implementar a interface ISomaPresenter, ela deverá codificar o que foi definido naquela estrutura. No caso do exemplo, deverão ser implementadas as propertys Model e View, juntamente com seus métodos de leitura e escrita (GetModel, SetModel, GetView e SetView).

Complementando, é necessário também a implementação do método Somar que, de certa forma, é o elemento principal neste contexto. A codificação destes métodos é apresentada na **Listagem 5**, onde nas linhas iniciais são apresentados os métodos de leitura e escrita utilizados pelas propriedades da classe. Em função destes métodos foram criados dois campos privados, FView e FModel, que serão usados para armazenar as instâncias de View (tipo ISomaView) e Model (tipo TSoma) a serem atribuídas e utilizadas internamente pela classe.

Listagem 4. Declaração da classe TSomaPresenter

```
type
 TSomaPresenter = class(TInterfacedObject, ISomaPresenter)
 private
  FView: ISomaView;
  FModel: TSoma;
  function GetModel: TSoma;
  procedure SetModel(const Value: TSoma);
  function GetView: ISomaView;
  procedure SetView(const Value: ISomaView);
 public
  property Model: TSoma read GetModel write SetModel;
  property View: ISomaView read GetView write SetView;
  function Somar: Integer;
 end;
```

Listagem 5. Implementação dos métodos classe TSomaPresenter

```
01 function TSomaPresenter.GetModel: TSoma;
02 begin
03   Result := FModel;
04 end;
05
06 function TSomaPresenter.GetView: ISomaView;
07 begin
08   Result := FView;
09 end;
10
11 procedure TSomaPresenter.SetModel(const Value: TSoma);
12 begin
13   FModel := Value;
14 end;
15
16 procedure TSomaPresenter.SetView(const Value: ISomaView);
17 begin
18   FView := Value;
19 end;
20
21 function TSomaPresenter.Somar: Integer;
22 begin
23   Model.Valor1 := View.Valor1;
24   Model.Valor2 := View.Valor2;
25   Result := Model.Somar;
26 end;
```

A partir daí, o destaque fica por conta da função Somar em que é feita uma chamada ao método Somar do modelo (linha 25), ao mesmo tempo em que os valores a serem utilizados na operação são lidos da View (linhas 23 e 24). Estas camadas (View e Model) são referenciadas na classe através das propertys de mesmo nome. Dessa forma, cabe somente atribuir a cada property, no momento da instanciação de um objeto desta classe, as devidas instâncias de objetos referentes à View e Model, como será visto ao final do exemplo.

Nota do DevMan

No Delphi, ao se declarar uma classe que implementa uma ou mais interfaces, é recomendável que esta descenda de TInterfacedObject. Esta classe nativa implementa uma interface também nativa, denominada IInterface. Esta interface possui a assinatura de três métodos, QueryInterface, _AddRef e _Release, que são implementados por TInterfacedObject. Através deles, TInterfacedObject faz o tratamento automático da contagem de referências e do gerenciamento de memória dos objetos.

View

Da mesma forma que foi definida a classe concreta que implementa a interface da camada Presenter, para a View também é necessária a definição da classe que irá implementar a interface da camada View definida neste exemplo como ISomaView.

Além disso, por se tratar da camada de visão da estrutura da aplicação, esta classe será codificada como sendo descendente do tipo TForm, provendo assim uma interação com o usuário. Na verdade, este é o formulário apresentado na **Figura 5**. Portanto, é através da View que ele poderá informar os valores a serem somados, e também através dela que o mesmo poderá visualizar o resultado da operação. A **Listagem 6** apresenta a declaração da classe.

Listagem 6. Declaração da classe TSomaView

```
TSomaView = class(TForm, ISomaView)
  ...
  {ocultada a declaração dos componentes e seus eventos}
 private
  FPresenter: ISomaPresenter;
  FValor2: Integer;
  FValor1: Integer;

  function GetPresenter: ISomaPresenter;
  procedure SetPresenter(const Value: ISomaPresenter);
  function GetValor1: Integer;
  procedure SetValor1(const Value: Integer);
  function GetValor2: Integer;
  procedure SetValor2(const Value: Integer);
 public
  property Presenter: ISomaPresenter read GetPresenter write SetPresenter;
  property Valor1: Integer read GetValor1 write SetValor1;
  property Valor2: Integer read GetValor2 write SetValor2;
  function MostrarView: TModalResult;
 end;
```

Conforme o "contrato" estabelecido pela interface ISomaView, esta classe implementa duas propertys, Valor1 e Valor2, que irão manipular os valores informados pelo usuário, deixando-as disponíveis à camada Presenter. Cada property faz uso de métodos de leitura e gravação (Get e Set) de seus valores, e assim como aconteceu na classe anterior, foram declarados campos privados (FValor1 e FValor2) para o armazenamento interno destes valores. A **Listagem 7** apresenta a implementação destes métodos Get e Set das propriedades.

GetValor1 e GetValor2 são os métodos que fazem a leitura dos valores das propertys Valor1 e Valor2, que será usada pela classe TSomaPresenter. Para simplificar o exemplo, e aproveitando que foi definida uma interface visual (um formulário) com os controles visuais referentes a cada valor (componentes TEdit), estes métodos farão a leitura direta dos valores digitados em cada componente, conforme pode ser visto nas linhas 3 e 8.

Os métodos de atribuição SetValor1 e SetValor2, recebem um valor como parâmetro, que são então armazenados nos campos privados FValor1 e FValor2, respectivamente, ao mesmo tempo que atualiza estes valores nos componentes visuais da interface gráfica. Apesar de não estarem sendo utilizados neste exemplo, tais métodos foram citados e codificados para que se mantenha a abordagem do exemplo mais próxima do mundo real.

Já a property Presenter é a que merece mais atenção neste momento, uma vez que é através dela que se estabelece o vínculo com a camada Presenter que, neste exemplo, trata-se da classe TSomaPresenter, mas por estar se referenciando uma interface, poderia ser qualquer classe que implemente a interface ISomaPresenter. Outro adentro a se fazer é que foi criado um campo privado (FPresenter) para o armazenamento desta instância de Presenter, conforme pode ser visto na declaração da classe.

Complementando a codificação da classe TSomaView, a **Listagem 8** apresenta a implementação do método MostrarView e do Click do botão, que funcionam como métodos de apoio à solução proposta. São ditos "métodos de apoio" visto que eles não são vitais para o ideal emprego do padrão MVP, todavia, são imprescindíveis para o bom funcionamento do sistema de exemplo que está se construindo.

Dito isto, o método MostrarView basicamente faz a chamada do método ShowModal do formulário, para que seja possível visualizá-lo em tela em runtime. Já o Click do botão utiliza-se de uma instância da camada Presenter (do tipo ISomaPresenter) para fazer a chamada ao método Somar e, em vista do resultado retornado pela função, este valor é exibido em tela pelo controle edtResultado. Por enquanto ainda não se apegue muito aos detalhes referentes à relação entre as camadas, visto que será abordado a seguir.

Executando o exemplo

Com toda a estrutura já pronta do sistema, basta colocá-la em ação. Neste momento é fundamental que as instâncias de cada módulo envolvido sejam devidamente criadas para o sucesso da execução da solução. Isto acontece até mesmo pela própria moldura do padrão em questão. Por exemplo, ao se instanciar um objeto View (TSomaView), sabe-se que no MVP este irá necessitar de uma instância de Presenter (TSomaPresenter), enquanto que esta, além da View, necessita se comunicar com uma instância de Model (TSoma).

Listagem 7. Implementação dos métodos Get e Set de TSomaView

```
01 function TSomaView.GetValor1: Integer;
02 begin
03   Result := StrToInt(edtValor1.Text);
04 end;
05
06 function TSomaView.GetValor2: Integer;
07 begin
08   Result := StrToInt(edtValor2.Text);
09 end;
10
11 function TSomaView.GetPresenter: ISomaPresenter;
12 begin
13   Result := FPresenter;
14 end;
15
16 procedure TSomaView.SetPresenter(const Value: ISomaPresenter);
17 begin
18   FPresenter := Value;
19 end;
20
21 procedure TSomaView.SetValor1(const Value: Integer);
22 begin
23   FValor1 := Value;
24   edtValor1.Text := IntToStr(Value);
25 end;
26
27 procedure TSomaView.SetValor2(const Value: Integer);
28 begin
29   FValor2 := Value;
30   edtValor2.Text := IntToStr(Value);
31 end;
```

Listagem 8. Implementação dos métodos restantes de TSomaView

```
function TSomaView.MostrarView: TModalResult;
begin
  Result := Self.ShowModal;
end;
procedure TSomaView.btnSomarClick(Sender: TObject);
begin
  edtResultado.Text := IntToStr(Presenter.Somar);
end;
```

Dessa forma, a chamada da View deste exemplo poderia ser feita conforme a **Listagem 9**. Das linhas 2 a 4, estão as variáveis de cada tipo necessário envolvido, representando as três camadas do padrão: Model, View e Presenter. Veja que, seguindo as boas práticas de programação, View e Presenter estão declaradas como sendo de um tipo abstrato (interface), dessa forma, qualquer classe que implemente tal interface poderá ser referenciada no código. Nas linhas 6, 7 e 8 ocorre a instanciação de cada classe referente a cada camada do sistema.

Retomando um pouco da teoria do MVP, sabemos que a camada Presenter deve ter conhecimento tanto de Model quanto de View, sendo assim, nas linhas 10 e 11 ocorrem a atribuição de ambas. Já a View deverá se relacionar com a Presenter, e essa relação é estabelecida na linha 13. A chamada ao método MostrarView da View exibe o formulário em tela. A **Figura 6** mostra o exemplo em funcionamento.

Mediante o que foi apresentado, um desenvolvedor mais experiente pode ter percebido que não está tão intuitivo instanciar as partes do sistema e fazer as devidas associações entre elas de forma independente. Uma abordagem visando a melhora deste tipo de situação seria a redefinição dos construtores das classes, de forma a exigir como argumento a instância do módulo necessário. Um exemplo seria passar um objeto Presenter à View, já no momento de sua criação, através de seu construtor.

O mesmo acontece com o objeto Presenter, que poderia exigir, no momento de sua instanciação, a passagem de um objeto Model e de um objeto View, tornando assim o trabalho do programador mais intuitivo.

Listagem 9. Chamadando a View do exemplo

```
01 var
02   Model: TSoma;
03   View: ISomaView;
04   Presenter: ISomaPresenter;
05 begin
06   View := TSomaView.Create(Self);
07   Presenter := TSomaPresenter.Create;
08   Model := TSoma.Create;
09
10   Presenter.Model := Model;
11   Presenter.View := View;
12
13   View.Presenter := Presenter;
14   View.MostrarView;
15 end;
```

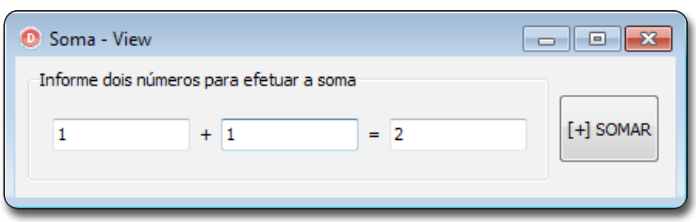


Figura 6. Aplicação de exemplo em funcionamento

Analisando o fluxo do sistema

Com o sistema em execução, uma forma de se entender melhor o que está acontecendo nos bastidores e assim compreender toda a interação entre as camadas do padrão MVP, é fazendo a depuração do aplicativo. Na chamada da View, vimos que ocorre a instanciação dos objetos relativos à Model, Presenter e a própria View. Com o formulário aberto, ao se preencher as caixas de entrada com valores e clicando no botão Somar, estamos efetivamente fazendo a estrutura que definimos até aqui funcionar.

O primeiro evento que ocorre é a chamada, por parte da View (TSomaView), do método Somar da camada Presenter. Isto porque a View não sabe nada sobre as regras de negócio do sistema, ou seja, ela não sabe como se dará a operação de soma, definida no modelo. Com o fluxo em Presenter (TSomaPresenter), o seu método Somar também não sabe dos detalhes da implementação deste método, ele somente pega os valores provindos da View e os repassa para a camada Model (TSoma) para, a seguir, fazer a chamada ao método Somar do próprio modelo. Com o fluxo agora em Model, o método Somar é efetivamente executado, retornando

o valor do resultado de volta à Presenter, que repassa para a View, onde este valor será exibido em tela.

Dessa forma, mediante um simples Debug, podemos ver claramente a separação de responsabilidades que ocorre entre as camadas, como a camada Presenter fazendo o "meio-de-campo" entre View e Model.

Conclusão

Para concluir este artigo, primeiramente não poderia deixar de prestar os devidos agradecimentos ao editor técnico desta revista, Paulo Quicoli, pois a parte prática deste exemplo foi inspirada em sua matéria publicada no Canal Delphi do Portal DevMedia, onde o mesmo aborda o padrão MVP. Esta, sem dúvida, foi uma das fontes usadas por mim para dar início à aplicação deste padrão em muitos de meus projetos pessoais, há dois anos.

Delphi XE - Trial
https://downloads.embarcadero.com/free/delphi

ASP.NET MVC – Página Oficial
http://www.asp.net/mvc

MSDN Magazine – WPF Apps With The Model-View-ViewModel Design Pattern
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

MVVM Foundation
http://mvvmfoundation.codeplex.com/

Por conseguinte, neste artigo foi apresentada uma visão geral dos três principais padrões de arquitetura presentes no mundo Delphi atualmente. Seja utilizando o Delphi tradicional ou o mais recente Delphi Prism, estes padrões tem como foco estabelecer uma ideal separação de responsabilidades em um sistema, bem como isolar ao máximo a sua camada de apresentação.

Conforme foi dito, o emprego deste tipo de padrão não é tão comum em projetos Delphi Win32, por N motivos e, justamente por isso, fica como dica a leitura de artigos passados desta revista que abordam o emprego do MVC em projetos desse tipo. Isso fará com que você quebre paradigmas e até comece a pensar em seus projetos de forma macro.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 5 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas além de participar do programa Beta Tester Delphi Prism, da Embarcadero.

Dê seu feedback sobre esta edição!

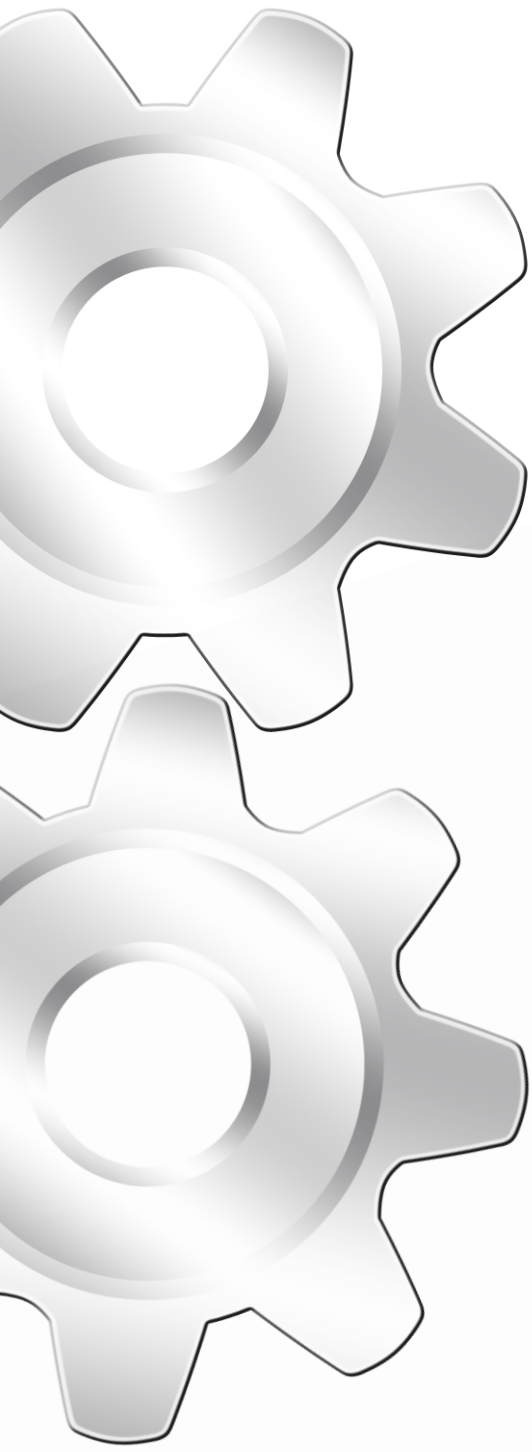
A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

POO - Dominando o uso de Interfaces

Vitor Luiz Rubio



Resumo DevMan

De que trata o artigo:

O artigo trata sobre interfaces, que são contratos estabelecidos para que classes os sigam. No artigo será visto como criar, usar, trocar e delegar interfaces. Também serão vistos erros comuns a se evitar e as novidades do Delphi XE para o tratamento de interfaces.

Em que situação o tema é útil:

Interfaces criam pontos de conexão entre partes de um sistema e ajudam a isolar do mundo exterior os detalhes de implementação de uma classe. Devem ser usadas sempre que se necessitam usar vários objetos semelhantes, mas que não são da mesma família, evitar ou corrigir o alto acoplamento, dividir tarefas entre programadores ou até mesmo entre consultorias terceirizadas e programar por "contratos". Elas são importantíssimas para criar sistemas com baixo nível de acoplamento e maior facilidade de manutenção. Programando-se orientado a interfaces facilita a distribuição das tarefas entre programadores e a criação de testes unitários. Além disso qualquer objeto pode ser substituído por qualquer outro que implemente a mesma interface. Isso proporciona um segundo nível de flexibilidade, pois permite que objetos de hierarquias distintas tenham algo em comum e possam ser intercambiados.

Interfaces:

Interfaces definem contratos ou pontos de conexão entre objetos. Se um método aceita como parâmetro uma interface A isso significa que ele aceita qualquer objeto que implemente essa interface. Da mesma forma um objeto que implemente uma interface B sempre pode ser substituído por qualquer objeto que implemente a mesma interface. Com o uso de interfaces é possível dizer de antemão qual é o comportamento esperado de um objeto e que métodos ele deve ter, forçando-o a cumprir esse contrato. Um gerente de projeto (ou coach ou scrum master dependendo da sua metodologia) pode dividir organizadamente tarefas entre programadores extraindo de cada estória de usuário, caso de uso, cartão CRC ou mesmo dos requisitos o comportamento básico de qualquer entidades. Até mesmo para a contratação de uma consultoria terceirizada é possível entregar uma interface e pedir: "quero uma classe ou componente que implemente essa interface". O "Como" a interface é implementada é "problema" de quem a implementa e o "que" se faz com ela é "problema" de quem a usa. Ao longo do artigo serão mostrados comparação, conversão implícita e explícita, operações com interfaces e alguns "minissistemas".

Interface é uma palavra com múltiplos significados, tanto na física, na comunicação e na ciência da computação. No caso da ciência da computação o termo foi "emprestado" mais de uma vez, pois pode significar o tipo ou padrão de conexão entre dois componentes, o conjunto de elementos gráficos para comunicar um estado ou aceitar comandos de um usuário (GUI) e um contrato ou padrão entre dois objetos ou componentes.

No caso do hardware, por exemplo, USB é uma interface. É possível conectar qualquer tipo de dispositivo na porta USB de um computador contanto que o dispositivo siga certos padrões, ou seja, implemente

a interface USB. O mesmo ocorre com PCI, PCI-express e outras tantas portas de conexão do computador.

Não é necessário ir muito longe: o TCP/IP fornece uma linguagem comum, acima dos protocolos físicos como ethernet, para que qualquer dispositivo em uma rede possa se comunicar em uma linguagem comum. O HTTP é um protocolo implementado sobre TCP/IP que recebe e envia dados através de requisições GET e POST e é a base de todas os sistemas online, redes sociais, webservices e computação em nuvem.

Qualquer objeto tem uma interface, mesmo que seja implícita, pois tem um conjunto de métodos e atributos públicos, que são visíveis e podem ser usados ou conectados com o meio exterior. A classe da **Listagem 1**, com seus métodos e atributos públicos e privados, tem uma interface implícita, que é o conjunto de seus membros públicos.

Nota

Todos os exemplos desse artigo foram feitos em um formulário com vários TButton, um para cada teste, e alguns métodos ou eventos. O código distribuído junto com esta edição tem mais exemplos além dos mostrados no artigo. O objetivo é que o leitor explore esse código modificando ou comentando partes para observar o que acontece.

Listagem 1. Classe de exemplo

```
TForm1 = class(TForm)
private
  { Private declarations }
public
  { Public declarations }
end;

TUmaClasse = class
private
  FUmaData: TDateTime;
  FUmNumero: integer;
  FUmNome: string;
  FUmaFracao: Double;
  FUmBooleano: boolean;
  procedure UmMetodoPrivado;
  property Data: TDateTime read FUmaData write FUmaData;
public
  procedure SetNome(const value: string);
  function GetNome: string;
  procedure DigaAlo;
  property Nome: string read GetNome write SetNome;
end;
```

Temos duas classes, um formulário e a classe TUmaClasse. Em TUmaClasse apenas SetNome, GetNome, DigaAlo e Nome são públicos, por isso eles definem uma interface implícita.

Em programação orientada a objetos quando se cria uma interface cria-se um contrato que deve ser respeitado por quem a implementa. A classe que implementa uma interface deve obrigatoriamente ter os métodos dela implementados.

Há uma certa similaridade com classes abstratas, mas a similaridade para por aí. É correto afirmar que uma classe totalmente abstrata não tem nenhum "comportamento" pois todo ele será criado nas classes derivadas, mas ainda assim uma classe derivada pode implementar alguns métodos e deixar outros ainda abstratos, para uma implementação futura. Adicionalmente uma classe abstrata ainda pode ter atributos públicos, ou propriedades, que podem ser herdados pelos seus descendentes. Uma classe totalmente abstrata não pode ser instanciada, mas uma classe que tem alguns métodos abstratos pode.

Além disso quando se começa uma família de classes a partir de uma classe abstrata todos os seus descendentes seguem uma herança linear, vertical. Usando interfaces esse cenário muda muito.

Para interfaces valem todas as regras de herança e polimorfismo, exceto que uma interface pode "herdar" de uma, duas ou mais interfaces. Quando uma interface herda de outra o termo técnico apropriado é "implementa" e não herda. O mesmo ocorre quando uma classe implementa uma interface.

Uma classe não pode ser descendente de uma interface, mas se for descendente de uma classe que implementa uma interface, ela e todos os seus descendentes passam automaticamente pela herança a implementar essa interface também.

Como múltiplas interfaces podem ser implementadas por uma classe ou interface é possível usar esse mecanismo para simular herança múltipla, fazendo com que todos os métodos de várias interfaces se combinem em uma única. Se a implementação das interfaces forem delegadas a objetos que já as implementem então tem-se herança múltipla "quase real" e com pouquíssimas linhas de código.

Um outro grande benefício das interfaces é que elas tem uma contagem de referências interna. Isso significa que elas não precisam ser destruídas, pois se auto destroem quando nenhum outro objeto precisa delas. Isso é o mais próximo que temos de um garbage collector.

Quando se programa orientado a interfaces automaticamente protege-se o software de uma dependência mútua de detalhes de implementação. Isso é um reforço para o conceito de encapsulamento, ou "caixa preta", pois um objeto que implementa uma interface pode ser substituído por qualquer outro que também implemente essa interface sem que o código pare de funcionar, mesmo que o comportamento do outro objeto seja totalmente diferente. Ou seja, se um objeto A implementa uma interface sua implementação deve ser isolada de qualquer objeto B que dependa (seja cliente, use) A. Desse modo, B usa apenas a interface de A, e qualquer alteração em A ou B pode ser feita contanto que esse contrato continue sendo respeitado.

No Windows, componentes DCOM, COM, COM+ e ActiveX implementam interfaces. A programação usando essa parte da API do Windows inevitavelmente usa ou implementa uma interface. Isso foi feito dessa maneira no Windows para que desenvolvedores pudessem criar, além de programas, extensões, plugins, complementos e suplementos para o próprio Windows, e também para o Word, Internet Explorer e assim por diante. Para isso, a partir da versão 3 do Delphi a linguagem mudou um pouco a hierarquia de interfaces para ser compatível com as interfaces COM do Win-

dows. Pelo mesmo motivo a partir da versão 3 o Delphi passou a suportar implementação múltipla e GUIDs. Nas versões 1 e 2 isso era impossível.

Nota do DevMan

O COM (Component Object Model) é uma tecnologia projetada pela Microsoft que atua tanto como especificação quanto implementação. Ela define como componentes e aplicações se comunicam, através de interfaces, e é a base para muitas outras tecnologias, tais como DCOM, MTS, ActiveX, e COM+. As tecnologias precursoras do COM+ foram o DCOM e o MTS. O DCOM permite distribuir objetos em uma rede, além de separar as regras de negócio em uma camada independente da aplicação cliente. O problema do DCOM é a escalabilidade. Ele não apresenta os mesmos resultados quando há um aumento do número de conexões, perdendo desempenho.

O MTS (Microsoft Transaction Server) incorporou diversos recursos ao DCOM. Um problema do DCOM era não possuir um gerenciador próprio. O MTS resolve o problema, fornecendo um gerenciador baseado no MMC (Microsoft Management Console), no qual podem ser configurados objetos e inúmeras opções.

O COM+ incorpora e estende o DCOM e o MTS e inclui o MSMQ (Microsoft Message Queue), herdando inúmeros recursos. As principais características do COM+ são gerenciamento, transações, escalabilidade, pooling e independência.

Ao escrever programas que suportem plugins considera-se o uso de interfaces, pois para escrever plugins para programas já existentes o conhecimento de interfaces é praticamente obrigatório.

É necessário perceber que o conceito de Interfaces, embora muito importante na programação orientada a objetos, não limita-se apenas às linguagens que suportam POO. Em um nível mais amplo, mais abstrato, interfaces são qualquer conjunto de métodos que um componente de software deve implementar para que seja compatível ou que possa trocar informações com outros componentes de software. Mesmo que esses métodos sejam estáticos, públicos e globais, como no caso da API do Windows, ainda persiste a ideia de "contrato". A própria sigla API é Application Programming Interface.

Geralmente APIs ou bibliotecas de classes e frameworks que contém interfaces têm manuais ou listas de interfaces que devem ser implementadas para garantir certas compatibilidades.

Interfaces devem ser vistas como o conjunto de dentes de uma engrenagem, os furos e pinos de peças de Lego ou ainda do conjunto de palavras e regras de uma linguagem humana.

Uma definição interessante para interfaces é que elas permitem você "interfacear" (neologismo horrível, reconheçamos) duas classes que não tem nada em comum. Por exemplo pode-se, em um formulário criado dinamicamente, definir uma interface em comum entre todos os tipos de componentes para se executar um mesmo método, ou obter uma mesma propriedade, em todos eles. Exemplos práticos de como fazer isso serão vistos mais adiante.

Interfaces no Delphi

Historicamente o conceito de interfaces vem da linguagem C, onde num arquivo de cabeçalho .h eram colocados as assinaturas, ou protótipos, dos métodos, mas os mesmos eram implementados apenas nos arquivos de código fonte .c.

O Delphi também "herdou" esse conceito do Pascal e do C. Em uma unit há duas seções principais, uma seção interface, onde coloca-se apenas os cabeçalhos ou protótipos das procedures e functions e a seção implementation onde coloca-se a implementação dos mesmos. As units, com suas seções de interface e implementação, eram o mais próximo que o Pascal podia chegar da POO numa época em que ele ainda não tinha efetivamente implementado todas as características e requisitos da POO.

Hoje o Delphi é totalmente compatível com orientação a objeto, mas herdou muitas características do Pascal, quer por questões de compatibilidade ou por motivos de elegância ou não descaracterização da linguagem. Como resultado disso a palavra "interface" no Delphi tem duplo significado: pode significar a seção interface de uma unit ou uma interface na visão da POO, que deve ser implementada por uma classe.

Como legado desse crescimento do Delphi e do Pascal em si, ainda hoje, colocamos o esqueleto de uma classe (também chamado de interface) na seção interface de uma unit, e a implementação dos métodos dessa classe na seção implementation.

Interfaces na prática se parecem muito com classes, mas não tem seus métodos implementados. O código a seguir mostra um exemplo de interface:

```
IVeiculo = interface
  procedure DeslocarSe;
end;
```

Não é possível falar sobre interfaces sem entrar no assunto programação orientada a objetos, já que os dois assuntos são relacionados e interdependentes. Na **Listagem 1** cabem dois adendos, duas particularidades do Delphi. A primeira é que os campos private podem ser acessados por outra classe se esta estiver na mesma unit, ou seja, eles podem ser acessados pelo formulário 1. Isso é uma quebra de encapsulamento que pode ser resolvida com a palavra chave strict antes de private, assim os campos privados não serão mais acessíveis pelo formulário 1.

Outra particularidade são as propriedades. Há um consenso entre praticantes da POO quanto ao lugar dos dados ou campos, que representam o estado de um objeto, e os métodos, que representam suas ações. Essa prática diz que todo atributo deve ser privado, e que deve ser acessado por métodos acessores. Os métodos para obter o valor do atributo são precedidos de "Get" e os métodos para modificar o atributo, caso ele não seja somente leitura, são precedidos de "Set". Esses métodos Get e Set são chamados de métodos acessores, e teoricamente todo atributo deveria ser privado e encapsulado por eles. Isso garante que você pode mudar, substituir, operar o atributo dentro da classe sem que ele seja exposto às outras classes e sem mudar a interface da classe. Isso mesmo, a preocupação principal é com a interface, para não quebrar sua conexão com outros objetos. Isso significa que um atributo Fnome do tipo string, enquanto estiver sendo controlado pelos acessores GetNome e SetNome, ele pode ser modificado para um label com os acessores usando seu caption, um TstringList com os acessores manipulando seu Text, pode ser quebrado e formatado na entrada e pode ter informações

concatenadas na saída sem que o Get e o Set sejam modificados, mantendo assim uma mesma interface, uma mesma propriedade string que não importa como ela é modificada internamente ela sempre será controlada da mesma forma pelos métodos Get e Set.

É aí que entra a particularidade do Delphi: com a palavra reservada property mais os modificadores read e write é possível criar uma propriedade Nome do tipo string para "ler" do método Get e "escrever" no método set, que na verdade corresponde à passagem de um parâmetro. Consequentemente os métodos acessores não necessariamente precisam ser públicos, eles podem ser privados e a propriedade que os usa ser pública. Isso é um segundo nível de encapsulamento na propriedade.

Muitos programadores do universo Java criticam essa funcionalidade do Delphi dizendo que isso é uma quebra de encapsulamento, pois expõe um campo privado, mas isso não é verdade. Se a propriedade estiver corretamente ligada a métodos Get e Set o campo privado está muito bem encapsulado, debaixo de duas camadas e o uso da propriedade é na verdade a execução de dois métodos, o Get para leitura e o Set para escrita. Não há violação de encapsulamento, não há exposição de campos privados, há apenas um "syntax sugar" para tornar o uso de uma propriedade PARECIDO com o uso de um atributo, quando na verdade é um método que está sendo executado.

Nesse ponto mora um problema: Algumas propriedades de muitas classes conhecidas e componentes da VCL não estão ligadas a métodos Get e Set e sim ligadas diretamente a campos privados. Isso sim é uma violação do encapsulamento, além de uma codificação desnecessária. Até pode-se argumentar que no futuro isso pode ser mudado para acessores Get e Set mas isso é especulativo. Por isso é altamente recomendável que sempre se use os métodos acessores e nunca se use algo como:

property Nome: **string read** FUmNome **write** FUmNome;

Tudo isso foi dito porque há mais uma peculiaridade do Delphi que deve ser explicada: como propriedades PODEM SER execução de métodos, interfaces podem ter propriedades apenas se estas estiverem ligadas a métodos Get e Set que sejam públicos. Construindo dessa forma seus objetos haverá dois "estilos" de uso do mesmo, ou usando a propriedade ou usando os métodos acessores, agradando assim usuários de Java e de Delphi, mas o resultado será o mesmo.

Uma interface "ISerVivo" tem basicamente os métodos para respirar e se alimentar. Uma interface "IAnimal" implementa a interface ISerVivo e tem o método "Locomover" fazendo de conta que todo animal se locomove de alguma forma.

Esse artigo não é sobre biologia e tampouco pretende ser um tratado exato sobre o assunto. Os exemplos aqui são meramente didáticos.

Uma interface ICavalo pode implementar IAnimal e adicionar o método relincha. Seguindo essa lógica um IVeiculo pode ser um veiculo genérico que tem o método "deslocarse". Interfaces que podem ser filhas de IVeiculo são ICarro, IBarco, ITrem e IAviao.

Na UML interfaces podem ser representadas por retângulos de classes sem atributos e com o estereótipo <<Interface>>. Quando uma classe requer uma interface ela pode ter o Receptor e quando ela provê uma interface ela pode ter o Conector (chamado de lollipop ou pirulito). A herança de interfaces é representada por uma seta de generalização, assim como em classes, e a implementação de interfaces é feita com uma seta pontilhada de Realização.

A **Listagem 2** mostra todas essas interfaces e classes que as implementam, e a **Figura 1** mostra como se representam interfaces na UML. As setas pontilhadas representam a implementação de uma interface, enquanto as setas sólidas representam uma herança.

Pode-se ver o uso das GUIDs, que são identificadores únicos das classes, formados por um par de chaves contendo 32 dígitos. Esse número é gerado e reconhecido por uma API do Windows, é uma identidade única que tem uma probabilidade muito ínfima de se repetir, a não ser que se use ctrl-c ctrl-v. Não é o programador que define a GUID, mas sim o Delphi. Pressionando ctrl+shift+g abaixo da declaração da interface uma GUID é gerada. As GUIDs não são característica intrínseca da orientação a objetos, mas são necessárias para a interoperabilidade com as tecnologias COM, COM+, ActiveX da Microsoft.

Caso esqueça de usar GUIDs, alguns métodos como o Supports, QueryInterface e GetInterface, não funcionarão. Caso use herança de interfaces as classes que implementam as interfaces filhas devem ter as interfaces ancestrais na lista de implementação ou ser descendentes de classes que as implementem, caso contrário Supports, QueryInterface e GetInterface funcionarão apenas com as interfaces filhas causando erros com as interfaces ancestrais. É por isso que Tbarco, Tcarro e Taviao tem Iveiculo na lista de implementação. Icachorro e Iboi devem ter também IserVivo e Ianimal.

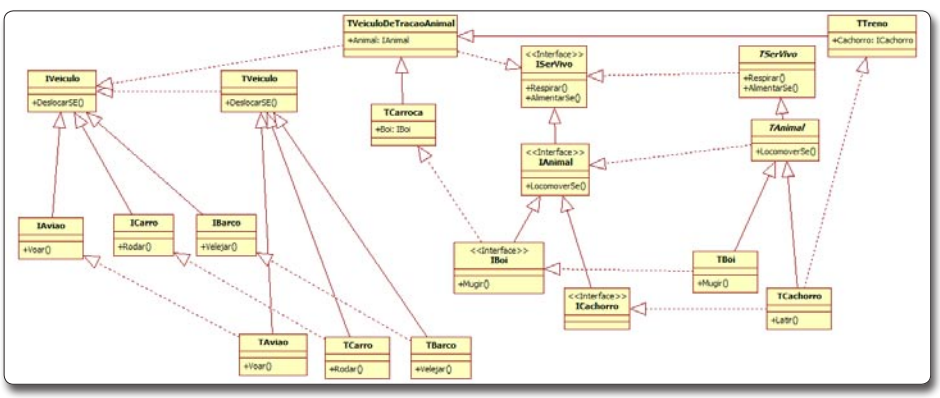


Figura 1. Modelo deste artigo em UML

Listagem 2. Interfaces e Implementações

```
IVeiculo = interface
['{3207B810-D5A8-4A72-97DE-005376FE7456}']
  procedure DeslocarSe;
end;
ICarro = interface(IVeiculo)
['{CB20C7C9-E042-49BA-BB2B-61F150ADFAE2}']
  procedure Rodar;
end;
IBarco = interface(IVeiculo)
['{D7D6AFC8-9005-488E-A8FA-1118F0F98D60}']
  procedure Velejar;
end;
IAviao = interface(IVeiculo)
['{26E9DB69-C58F-498B-AEAB-76147715BFE3}']
  procedure Voar;
end;
ISerVivo = interface
['{328D4469-7A22-4ED2-934A-3BACD963FA0D}']
  procedure Respira;
  procedure AlimentaSe;
end;
IAnimal = interface(ISerVivo)
['{BC22C426-37B3-42C8-8ADE-59E6F9AFB21F}']
  procedure LocomoverSe;
end;
TCachorro = class(TInterfacedObject, Icachorro, Ianimal, ISerVivo)
public
  //implementação de ISerVivo
  procedure Respira;   virtual;
  procedure AlimentaSe; virtual;
  //implementação de IAnimal
  procedure LocomoverSe; virtual;
  //implementação de Icachorro
  procedure Latir;   virtual;
end;
TBoi = class(TInterfacedObject, IBoi, Ianimal, ISerVivo)
public
  //implementação de ISerVivo
  procedure Respira;   virtual;
  procedure AlimentaSe; virtual;
  //implementação de IAnimal
  procedure LocomoverSe; virtual;
  //implementação de Iboi
  procedure Mugir;     virtual;
end;
TAviao = class(TInterfacedObject, IAviao, IVeiculo)
public
  procedure DeslocarSe;
  procedure Voar;
end;
TBarco = class(TInterfacedObject, IBarco, IVeiculo)
public
  procedure DeslocarSe;
  procedure Velejar;
end;
TCarro = class(TInterfacedObject, ICarro, IVeiculo)
public
  procedure DeslocarSe;
  procedure Rodar;
end;
{...}
{ TBoi }
procedure TBoi.AlimentaSe;
begin
  ShowMessage('Comendo Capim');
end;
procedure TBoi.LocomoverSe;
begin
  ShowMessage('Boi Andando pesadamente');
end;
procedure TBoi.Mugir;
begin
  ShowMessage('Muuuuuuuuuuuuuuuuuu');
end;
procedure TBoi.Respira;
begin
  ShowMessage('Boi Respirando');
end;
{ TCachorro }
procedure TCachorro.AlimentaSe;
begin
  ShowMessage('Comendo Ração');
end;
procedure TCachorro.Latir;
begin
  ShowMessage('Au Au');
end;
procedure TCachorro.LocomoverSe;
begin
  ShowMessage('Cachorro Andando');
end;
procedure TCachorro.Respira;
begin
  ShowMessage('Cachorro Respirando');
end;
{ TAviao }
procedure TAviao.DeslocarSe;
begin
  ShowMessage('Avião deslocando-se');
end;
procedure TAviao.Voar;
begin
  ShowMessage('Voando');
  DeslocarSe;
end;
{ TBarco }
procedure TBarco.DeslocarSe;
begin
  ShowMessage('Barco Deslocando-se');
end;
procedure TBarco.Velejar;
begin
  ShowMessage('Velejando');
  DeslocarSe;
end;
{ TCarro }
procedure TCarro.DeslocarSe;
begin
  ShowMessage('Carro Deslocando-se');
end;
procedure TCarro.Rodar;
begin
  ShowMessage('Rodando');
  DeslocarSe;
end;
```

Essas podem ser omitidas caso não se deseje trabalhar com os métodos Supports, QueryInterface e GetInterface.

Ao se utilizar interfaces não é necessário se preocupar com a destruição das mesmas, devido a contagem de referências, e consequentemente reduz-se o risco de memory leaks. Entretanto algumas regras devem ser obedecidas. Uma dessas regras é que nunca se deve misturar interfaces e objetos. Veremos exemplos que explicarão porque.

Uma classe seria usada apenas na hora de se criar o objeto, usando-se o construtor, mas todas as ligações com outros objetos, chamadas a métodos e passagem de parâmetros deve ser feita com uma "variável" do tipo da interface, e não do tipo do objeto. Isso pode ser visto na **Listagem 3**.

Listagem 3. Procedure que aceita um IVeiculo

```
procedure TForm1.FazVeiculoAndar(veiculo: IVeiculo);
begin
 veiculo.DeslocarSe;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
 aviao: IAviao;
 carro: ICarro;
 barco: IBarco;
begin
 aviao:= TAviao.Create;
 carro:= TCarro.Create;
 barco:= Tbarco.Create;

 FazVeiculoAndar(aviao);
 FazVeiculoAndar(carro);
 FazVeiculoAndar(barco);
```

As variáveis aviao, carro, barco são dos tipos Iaviao, Icarro e Ibarco, respectivamente. Repare que o método FazVeiculoAndar deve aceitar um Iveiculo como argumento, mesmo assim todas essas chamadas funcionaram.

Uma particularidade importante a ser notada aqui: a implementação de Taviao foi feita da seguinte maneira: TAviao = class(TInterfacedObject, IAviao, IVeiculo). Não seria necessário colocar Iveiculo na lista de implementação já que Iaviao é descendente de Iveiculo, portanto ele contém todos os métodos de Iveiculo e está implícito que Iaviao também é Iveiculo. O mesmo vale para Tbarco e Tcarro. O objetivo de colocar Iveiculo na lista de implementação é para que funcione o método Supports.

O método Supports é usado para saber se um objeto suporta uma interface, e para poder fazer typecasts. Já foi mostrado que uma variável do tipo interface recebe um tipo objeto que implemente essa interface, já que é sabido que o método Create de um objeto retorna uma instância do mesmo. Isso quer dizer que variáveis do tipo objeto podem ser atribuídas a variáveis do tipo interface caso a implementem. Caso não implementem um erro será disparado, então o código a seguir está errado:

```
var
 errado: IAviao;
 outroerrado: IVeiculo;
begin
 errado := Tbarco.Create;
 outroerrado := TUmaClasse.Create;
```

Já a conversão de qualquer um desses veículos para a interface base deles, Iveiculo, é correta. A **Listagem 4** mostra como objetos podem ser atribuídos a interfaces e como qualquer um dos três veículos do exemplo implementa Iveiculo.

Listagem 4. Veículos usados como IVeiculo

```
procedure TForm1.Button3Click(Sender: TObject);
var
 aviao: IAviao;
 carro: ICarro;
 barco: IBarco;
 qualquerveiculo: IVeiculo;

 objaviao: TAviao;
 objCarro: TCarro;
 objBarco: TBarco;
begin
 objaviao:= TAviao.Create;
 objCarro:= TCarro.Create;
 objBarco:= Tbarco.Create;

 aviao := objaviao;
 carro := objCarro;
 barco := objBarco;
 qualquerveiculo := Tbarco.Create;


 FazVeiculoAndar(aviao);
 FazVeiculoAndar(carro);
 FazVeiculoAndar(barco);
 FazVeiculoAndar(qualquerveiculo);
```

A **Listagem 5** mostra conversões de tipo entre interface/objeto e a **Listagem 6** as conversões objeto/interface, bem como o uso correto dos operadores "is" e "as".

A primeira coisa que se percebe na **Listagem 5** é que para usar o operador "as" para saber se um objeto é de uma determinada interface esse objeto deve ser obrigatoriamente do tipo TInterfacedObject ou de um objeto que implemente uma interface. A segunda coisa é que um tipo objeto sempre pode ser convertido para o tipo da interface que ele implementa.

O operador "IS" não pode ser usado para saber se um objeto é de um tipo de interface, mas pode ser usado para saber se uma variável interface é do tipo de alguma classe. Ou seja, do lado esquerdo do IS podem haver instâncias de objetos em variáveis do tipo interface ou objeto, mas do lado direito apenas classes, nunca funcionará com Interfaces.

Conversão de tipo objeto para interface pode ser usado com o operador "As", mas com o cuidado de colocar isso dentro de um try...except, pois o objeto passado pode não implementar a interface para a qual se está tentando converter. Se isso acontecer

Listagem 5. Conversões implícitas e explícitas de objeto para interface

```
procedure TForm1.Button3Click(Sender: TObject);
var
  aviao: IAviao;
  carro: ICarro;
  barco: IBarco;
  objAviao: TAviao;
  objCarro: TCarro;
  objBarco: TBarco;
  UmObjetoQualquer: TInterfacedObject;

  OutroVeiculo: IVeiculo;
  OutroCarro: ICarro;
  OutroBarco: IBarco;
  UmSerVivo: ISerVivo;

begin

  objaviao:= TAviao.Create;
  objCarro:= TCarro.Create;
  objBarco:= Tbarco.Create;
  UmObjetoQualquer := TCarro.Create;

  //conversão objeto/interface
  //pode ser o constructor aqui, mas não se deve misturar
  aviao := objaviao;
  carro := objCarro;
  barco := objBarco;

  aviao.Voar;
  carro.Rodar;
  barco.Velejar;

  FazVeiculoAndar(objaviao);
  FazVeiculoAndar(objCarro);
  FazVeiculoAndar(objBarco);

  //não fazer isso caso os objetos foram instanciados em variáveis do tipo
  //objeto e não existe uma variavel interface apontando para eles.
  //Isso pode zerar o contador de referências e destruir o objeto
  (objAviao as IAviao).Voar;
  (objCarro as ICarro).Rodar;
  (objBarco as IBarco).Velejar;
  (UmObjetoQualquer as ICarro).Rodar;


  if Supports(UmObjetoQualquer.ClassType, IVeiculo) then
  begin
    ShowMessage('UmObjetoQualquer suporta a interface IVeiculo');
    (UmObjetoQualquer as IVeiculo).DeslocarSe;
    FazVeiculoAndar(UmObjetoQualquer as IVeiculo);
    OutroVeiculo := (UmObjetoQualquer   as IVeiculo);
    OutroVeiculo.DeslocarSe;
  end;

  if Supports(UmObjetoQualquer.ClassType, ICarro) then
  begin
    ShowMessage('UmObjetoQualquer suporta a interface ICarro');
    (UmObjetoQualquer as ICarro).DeslocarSe;
    FazVeiculoAndar(UmObjetoQualquer as ICarro);
    OutroVeiculo := (UmObjetoQualquer as ICarro);
    OutroVeiculo.DeslocarSe;
  end;

  if Supports(UmObjetoQualquer, IVeiculo, OutroVeiculo) then
  begin
    ShowMessage('UmObjetoQualquer suporta a interface IVeiculo');
    FazVeiculoAndar(OutroVeiculo);
  end;

  if Supports(UmObjetoQualquer, ICarro, OutroVeiculo) then
  begin
    ShowMessage('UmObjetoQualquer suporta a interface ICarro');
    FazVeiculoAndar(OutroVeiculo);
  end;

  if (aviao is TAviao) then
  begin
    ShowMessage('Avião é um TAviao');
    (aviao as TAviao).DeslocarSe;
  end;

  if (carro is TCarro) then
  begin
    ShowMessage('Carro é um TCarro');
    (carro as TCarro).DeslocarSe;
  end;

  if (barco is TBarco) then
  begin
    ShowMessage('barco é um TBarco');
    (barco as TBarco).DeslocarSe;
  end;

  //sabe-se que UmObjetoQualquer é um  ICarro
  //usando operador "as" para typecast

  try
    OutroCarro := nil;
    OutroCarro := (UmObjetoQualquer as ICarro);
  except
    on EIntfCastError do
      ShowMessage('Interface Errada');
  end;
  if OutroCarro <> nil then
    OutroCarro.Rodar;

  try
    OutroBarco := nil;
    OutroBarco := (UmObjetoQualquer as IBarco);
  except
    on EIntfCastError do
      ShowMessage('Interface Errada');
  end;
  if OutroBarco <> nil then
    OutroBarco.Velejar;


  if carro.QueryInterface(IVeiculo,OutroVeiculo) = S_OK  then
    OutroVeiculo.DeslocarSe;

  if barco.QueryInterface(IVeiculo,OutroVeiculo) = S_OK  then
    OutroVeiculo.DeslocarSe;

  if aviao.QueryInterface(IVeiculo,OutroVeiculo) = S_OK  then
    OutroVeiculo.DeslocarSe;

  if UmObjetoQualquer.GetInterface(IVeiculo, OutroVeiculo) then
    OutroVeiculo.DeslocarSe;

  if UmObjetoQualquer.GetInterface(ISerVivo, UmSerVivo)  then
    UmSerVivo.AlimentaSe;

end;
```

Listagem 6. Conversões de interface para objeto

```
procedure TForm1.Button4Click(Sender: TObject);
var
  aviao: IAviao;
  carro: ICarro;
  barco: IBarco;
  outroObjAviao: TAviao;
  outroObjCarro: TCarro;
  outroObjBarco: TBarco;
  UmInterfacedObject: TInterfacedObject;
  UmObjeto: TObject;
begin

  aviao:= TAviao.Create;
  carro:= TCarro.Create;
  barco:= Tbarco.Create;

  //conversão interface/objeto com AS
  if aviao is TAviao then
  begin
    ShowMessage('Avião é um TAviao');
    outroObjAviao := (aviao as TAviao);
    outroObjAviao.Voar;
  end;

  //conversão interface/objeto com AS
  if carro is TCarro then
  begin
    ShowMessage('Carro é um TCarro');
    outroObjCarro := (carro as TCarro);
    outroObjCarro.Rodar;
  end;


  //conversão interface/objeto com AS
  if barco is Tbarco then
  begin
    ShowMessage('barco é um Tbarco');
    outroObjbarco := (barco as Tbarco);
    outroObjbarco.Velejar;
  end;


  //conversão errada
  outroObjbarco := nil;
  try
    outroObjbarco := (carro as Tbarco);
  except
    on EInvalidCast do
      ShowMessage('Tipo Errado');
  end;
  if outroObjbarco <> nil then
    outroObjBarco.Velejar;


  if carro is TInterfacedObject then
  begin
    ShowMessage('carro é um TInterfacedObject');
    UmInterfacedObject := (carro as TInterfacedObject);
    ShowMessage('Classe: ' + UmInterfacedObject.ClassName);
  end;

  if carro is TObject then
  begin
    ShowMessage('carro é um TObject');
    UmObjeto := (carro as TObject);
    ShowMessage('Classe: ' + UmObjeto.ClassName);
  end;

end;
```

uma exception do tipo EintfCastError será disparada. O operador "as" pode ser usado inclusive para os tipos primitivos das interfaces, contanto que esses tipos ancestrais sejam implementados por uma classe ancestral, abstrata ou não, ou que estejam na lista de interfaces implementadas da classe.

Também é mostrado na **Listagem 5** o uso do método Supports. O método Supports serve para saber se um objeto qualquer, uma classe, uma instância ou uma interface suportam, ou implementam, um tipo determinado. Usando o IDE do Delphi percebe-se que existem 5 overloads do método Supports:

```
function Supports(const Instance: IInterface; const IID: TGUID; out Intf): Boolean;
overload;
function Supports(const Instance: TObject; const IID: TGUID; out Intf): Boolean;
overload;
function Supports(const Instance: IInterface; const IID: TGUID): Boolean; overload;
function Supports(const Instance: TObject; const IID: TGUID): Boolean; overload; //
Unsafe
function Supports(const AClass: TClass; const IID: TGUID): Boolean; overload;
```

Os métodos Supports dividem-se basicamente em dois tipos: um que aceita dois argumentos e um que aceita um argumento de saída. Os tipos que aceitam dois argumentos servem para saber se um objeto implementa uma interface. O tipo que aceita três parâmetros o último parâmetro é um de saída, marcado com "out". Se o resultado de Supports for verdadeiro e o objeto realmente suportar a interface então o parâmetro de saída será uma variável do tipo interface já com a referência ao objeto propriamente dito (o objeto de entrada). Caso contrário o parâmetro de saída será nil.

O método número 4 está marcado como unsafe em um comentário, por isso não deve ser usado nunca. Porém ele deve ser observado para que o leitor não cometa sem perceber o mesmo erro que esse método comete. O método Supports inseguro e seu comentário pode ser visto no código a seguir:

```
function Supports(const Instance: TObject; const IID: TGUID): Boolean;
var
  Temp: IInterface;
begin
  // NOTE: Calling this overload on a ref-counted object that has REFCOUNT=0
  // will result in it being freed upon exit from this routine.
  Result := Supports(Instance, IID, Temp);
end;
```

Por esse motivo na **Listagem 5** foi usado o método Supports que aceita uma classe como parâmetro e foi usado UmObjetoQualquer.ClassType para obter uma referência à classe de UmObjetoQualquer.

Nota do DevMan

O motivo do método Supports que aceita uma instância e um tipo interface como parâmetro acabar destruindo o objeto está escrito em um comentário do próprio método, no source da unit SysUtils: Se você converte um objeto para uma interface temporária o objeto é destruído assim que a interface não é mais usada, referenciada ou a execução sai do escopo onde a conversão foi feita.

Essa é uma das consequências de se misturar objetos com interfaces inadvertidamente. Sempre que se usa interfaces deve-se executar o Create para criar os objetos diretamente na variável interface.

Caso se atribua variáveis do tipo objeto em uma variável do tipo interface o objeto não pode nunca ser liberado com free, e não pode nunca ser usado fora desse escopo. Se alguém deve ser um atributo público ou privado da classe é a variável do tipo Interface, fora do escopo dos métodos e das conversões, e nunca a variável tipo objeto.

Na **Listagem 5** a linha (UmObjetoQualquer as ICarro).DeslocarSe cria uma referência a uma interface Icarro durante a conversão que não existirá mais depois da conversão, depois da execução do método, o que fará com que o objeto UmObjetoQualquer seja destruído.

O mesmo ocorre com a linha OutroVeiculo := (UmObjetoQualquer as ICarro) porém usando uma variável local no método: OutroVeiculo.

Essas duas linhas causariam erros inesperados caso UmObjetoQualquer fosse um field ou propriedade do formulário ou de uma classe qualquer, fora do método onde ocorrem as conversões. É o mesmo tipo de erro que o Supports causa.

O método Supports retornará true apenas se a interface estiver na lista de implementação da classe, então interfaces ancestrais, como Iveiculo, se não estiverem na lista serão ignoradas.

Existe outro tipo de verificação e conversão com o método QueryInterface. QueryInterface tem um objetivo duplo: fazer typecasts seguros entre uma instância de um objeto e uma interface que ele implemente ou verificar se ele implementa essa interface. QueryInterface pode ser usado para verificar se uma instância de Tcarro implementa ou "segue" Icarro.

O método QueryInterface aceita dois parâmetros, o primeiro é de entrada e é um tipo Interface. O segundo é de saída e é uma variável do tipo passado no primeiro argumento. UmaInstancia.QueryInterface(IminhaInterface, UmaVariavel) verificará se o objeto UmaInstancia também implementa IminhaInterface. Se for verdadeiro o método retornará S_OK, que tem valor zero, e a variável UmaVariavel será diferente de nil apontando para o mesmo objeto que UmaInstancia. Caso UmaInstancia não suporte IminhaInterface então o resultado será diferente de S_OK, podendo ser E_NOINTERFACE ou qualquer outro erro diferente de zero. Nesse caso UmaVariavel será nil e não deverá ser usada, pois não aponta para nenhum objeto.

Não deveria ser obrigatório uma classe implementar uma interface se ela já implementa um descendente dessa interface. Não faz sentido Tcarro implementar Iveiculo se ele já implementa Icarro, que contém Iveiculo implicitamente. Porém Supports, QueryInterface e GetInterface não funcionarão se a lista de interfaces que a classe implementa não incluir as interfaces ancestrais.

Existe uma maneira de contornar isso: Para que os métodos Supports, QueryInmterface e GetInterface funcionem corretamente por toda uma hierarquia de interfaces deve-se criar classes abstratas que sejam ancestrais das classes concretas Tcarro, Tbarco, TAviao, Tboi, TCachorro e que implementem as interfaces ancestrais Iveiculo, IserVivo e Ianimal, respectivamente. No código disponível você encontra as linhagens de Iveiculo e IserVivo separadas em units diferentes e com classes abstratas implementando as interfaces base.

Nota do DevMan

As versões anteriores do Delphi aceitavam a palavra abstract apenas em métodos virtuais. As versões novas, como a XE, aceitam a palavra abstract na declaração de uma. Isso protege contra a criação indevida de classes abstratas, que não devem ser instanciadas.

Uma classe marcada como abstrata avisa o programador caso ele a instancie por engano com uma Warning do tipo "[DCC Warning] uPrincipal.pas(408): W1020 Constructing instance of 'TUmaClasseAbstrata' containing abstract method 'TUmaClasseAbstrata.TesteAbstract'".

Mesmo assim isso não protege a classe de ser instanciada, pois a warning poderia ser ignorada. Objetos de classes abstratas devem ser usados para conter referências para instâncias de qualquer classe concreta derivada delas, mas classes abstratas devem ser instanciadas.

Para proteger definitivamente uma classe abstrata contra criação pode-se criar um Constructor virtual e abstract. Qualquer tentativa de criar uma instância desta classe resultará na exception "Abstract Error".

Nas versões anteriores do Delphi era possível converter uma variável "instância" para uma variável "interface". O contrário não era verdadeiro, os operadores IS e AS não funcionavam e a única maneira de fazer a conversão era com typecast forçado, com a obrigatoriedade de saber que tipo ou classe de objeto era aquela interface, caso contrário aconteceriam erros, access violations e memory leaks.

Como mostrado nas **Listagens 5** e **6** o operador AS pode ser usado para typecasts do tipo objeto/interface e graças à Embarcadero agora é possível, no Delphi XE, a conversão de interface para objeto com o operador AS, tornando-se assim um caminho de duas mãos. Infelizmente o operador IS ainda não pode ter um tipo interface à direita.

Na **Listagem 6** o operador "is" é usado com classes a direita, verificando que tipo de objeto uma variável interface carrega. O operador "as" é usado para fazer conversão segura Interface/objeto, coisa que não era possível em versões anteriores do Delphi a não ser que fossem feitas conversões forçadas.

O operador "as" pode ser usado também dentro de um try... except no caso do risco de se receber um objeto errado, como o exemplo em que se tenta converter a variável carro para um objeto do tipo Tbarco.

Finalmente uma interface pode ser convertida para um objeto genérico e através desse objeto pode-se usar RTTI para inferir que tipo de classe esse objeto é. Os dois últimos exemplos da **Listagem 6** convertem carro para TInterfacedObject ou TObject e mostram de que classe ele é com ShowMessage('Classe: ' + UmObjeto.ClassName). O resultado é 'Tcarro'.

Considerando agora um veículo movido por tração animal, como um Tcarroca ou Ttreno é necessário que ele implemente a interface de Iveiculo e de Ianimal ao mesmo tempo.

Além disso um Ttreno só pode ser puxado por Icachorros, enquanto que uma Tcarroca pode ser puxada por Icavalos ou Ibois. Adicionalmente, uma Tcarroca É um veículo, mas não se pode dizer que ela é um boi. Ela CONTÉM um boi. Isso pode ser feito através da delegação (**Listagem 7**): uma Tcarroca implementa Iveiculo mas implementará Iboi apenas se houver um objeto que já implemente Iboi associado.

Listagem 7. Delegação

```
TTreno = class(TVeiculoTracaoAnimal, ICachorro)
private
  FAnimal: ICachorro;
  function GetAnimal: ICachorro;  reintroduce;
  procedure SetAnimal(const value: ICachorro);   reintroduce;
public
  constructor Create(const OAnimal: ICachorro); reintroduce;
  //métodos próprios
  procedure Andar;  override;
  //métodos de IVeiculo
  procedure DeslocarSe; override;
  //não é necessário implementar
  property Animal: ICachorro read GetAnimal write SetAnimal
  implements ICachorro;
end;
```

TTreno não é um ser vivo, tampouco um cachorro, mas ele implementa a interface ICachorro através de um objeto do tipo ICachorro que ele carrega. Ou seja, um trenó deve TER um ICachorro e não SER um ICachoro. Ele delega todas as operações de ICachorro para um objeto TCachorro se existir.

Isso é delegação: uma classe implementar uma interface não nela mesma mas através de uma propriedade associada a ela. Há duas maneiras de associar um IBoi a uma TCarroca, uma delas é criar TCarroca como objeto, atribuir à variável interface e usar o objeto para associar o IBoi. Outra forma seria criar tanto carroca como boi em interfaces e converter carroca para objeto apenas para associar o IBoi.

Mas a melhor maneira seria se o boi fosse passado no construtor da classe. Nesse caso passando nil no construtor teria-se uma carroça sem boi, ou um trenó sem cachorro para puxar.

Caso a carroça não tenha boi, todos os métodos ou conversões que exigem a existência de um IBoi, IAnimal ou ISerVivo falharão com um erro de "Falta um boi para puxar essa carroça" ou "interface not supported" do tipo EintfCastError.

No uso do Supports ou operador "as" pode-se atribuir carroça à um boi, um animal ou um ser vivo caso ela tenha um boi. Mas a comparação ou cast deve ser feita sempre com IBoi (ou ICachorro, se for TTreno) caso contrário resultará em abstract error. Isso acontece porque a classe que implementa diretamente essas interfaces é abstrata. A conversão acontece para um TAnimal, ou TSerVivo, quando na verdade deveria ser para TBoi ou TCachorro.

Outra regra a ser seguida diz respeito à hierarquia: assim como o ancestral comum de todos os objetos é TObject, o ancestral comum de todas as interfaces é IInterface. Há um problema bem evidente aí: TObject não implementa os métodos básicos de IInterface.

A classe que implementa IInterface é TInterfacedObject. Qualquer classe que implemente qualquer interface deve ser descendente direto ou indireto de TInterfacedObject. Em consequência disso qualquer descendente de TInterfacedObject já implementa a interface base IInterface.

No entanto há muitas classes nativas da VCL que não são descendentes de TInterfacedObject. A maioria das classes na verdade não é dessa linhagem. Sendo assim, sempre que surgir a necessidade de criar um descendente de uma classe qualquer que implemente uma interface os métodos básicos de IInterface deverão ser implementados. Felizmente é possível ler o fonte de TInterfacedObject e se basear nele, embora isso possa criar código repetitivo.

Prática com componentes

Uma das utilidades de interfaces é poder usar várias classes diferentes de uma mesma maneira por fazê-las implementar uma mesma interface, com um mesmo conjunto de métodos.

Isso permite que uma mesma operação possa ser executada sobre objetos diferentes, e uma mesma informação obtida deles.

Quando se programa para Web, em qualquer linguagem, os diferentes controles de um formulário HTML, os inputs, chegam como variáveis ao servidor em uma espécie de hashlist. Todos eles podem ter seu "valor" obtido na forma de string. E se isso fosse possível com formulários Win32? Com interfaces é possível simular isso.

Foram criados descendentes dos componentes TEdit, TCheckBox, TMemo, TRadioButton, TRadioGroup que implementam a interface IControle.

Esta interface IControle tem um método chamado ObtemValor. A ideia é que não importa qual o componente em um formulário ele terá um conteúdo textual que pode ser obtido com ObtemValor, como mostra o exemplo da **Listagem 8**.

Esta unit deve ser colocada em um package e instalada, sendo os componentes registrados com RegisterComponents, mas para efeito de didática e agilidade ela será usada diretamente pelo formulário de teste desse artigo e os componentes serão criados e adicionados ao formulário em runtime, na criação do mesmo. O resultado será como mostra a **Figura 2**.

O código para varrer os controles do formulário e obter o seu valor pode ser visto na **Listagem 9**. Além do método ObtemValor foram adicionados os métodos Nome e Classe, que servem para obter o nome e a classe do componente.

Ao varrer os controles do formulário com um laço for é verificado, com a ajuda de Supports, se o controle implementa IControle. Isso é feito pois não se sabe com que controle está se lidando, já que o podem existir controles que não implementam IControle. Se o Supports retornar verdadeiro então usa-se a

variável "controle" para extrair as informações no memo.

Deve ser observado que os controles são diferentes e de hierarquias totalmente diferentes, mas o uso de interfaces faz com que todos eles possam ser usados da mesma maneira e que tenham métodos em comum. Propriedades do tipo boolean, como checked, são convertidas para string, propriedades do tipo integer, como ItemIndex no RadioGroup, são usadas para obter o item correspondente e seu texto.

Listagem 8. Componentes que implementam uma interface

```
 TCDEdit = class (TEdit, IControle)
   function Nome: string;
   function Classe: string;
   function ObtemValor: string;
 end;

implementation

{ TEditHelperInterfaced }

function TCDEdit.Classe: string;
begin
  Result := ClassName;
end;

function TCDEdit.Nome: string;
begin
  Result := Name;
end;

function TCDEdit.ObtemValor: string;
begin
  Result := Text;
end;
```

Listagem 9. Varrendo os componentes do formulário

```
procedure TForm1.Button6Click(Sender: TObject);
var
  i: integer;
  controle: IControle;
begin

  for i := 0 to ControlCount -1 do
    if Supports(Controls[i], IControle, controle) then
      mControles.Lines.Add(
        'Nome: ' + controle.Nome + #13#10 +
        'Classe: ' + controle.Classe + #13#10 +
        'Valor: ' + controle.ObtemValor
      );
end;
```

A lista de controles é varrida aqui com o vetor Controls e ControlCount pois ela não traz nenhum controle não visual ou que não seja descendente de TControl. Pode ser usado também Components e ComponentCount, mas o número de iterações do loop seria maior. Pode-se usar até mesmo FindComponent para encontrar um componente e verificar se ele é um IControle com Supports. Se executado depois de alguns controles editados o memo mControles terá o texto a seguir:

```
Nome: cbEmbrulharPresente
Classe: TCDCheckBox

Valor: True

Nome: cdrbCartao1
Classe: TCDRadioButton
Valor: Usar Visa = True

Nome: cdrbCartao2
Classe: TCDRadioButton
Valor: Usar Master = False

Nome: edNumCartao
Classe: TCDEdit
Valor: 123456789

Nome: mObservacao
Classe: TCDMemo
Valor: um teste qualquer

Nome: rgPagamento
Classe: TCDRadioGroup
Valor: 3 x
```

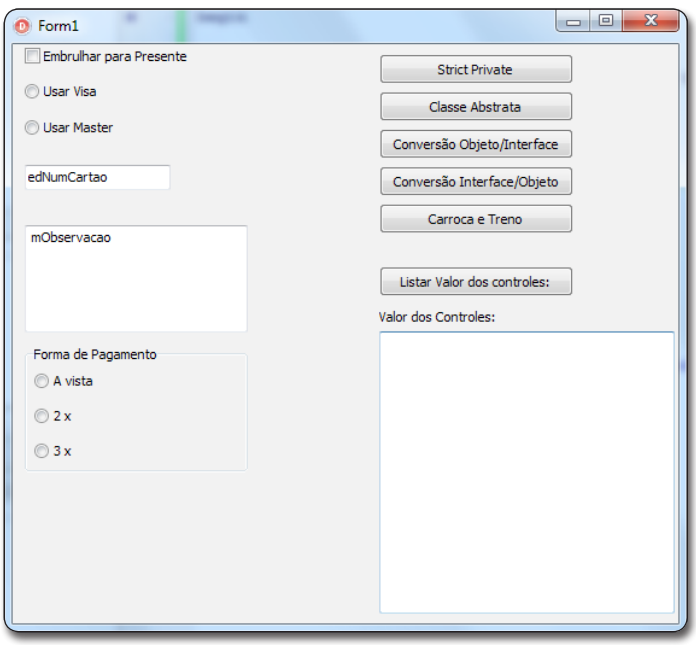


Figura 2. Formulário principal

Estendendo-se a interface IControle para conter propriedades comuns de componentes, Getters e Setters pode-se usar qualquer componente em lugar de qualquer outro. Além disso pode-se usar um método para obter um controle por nome e o seu valor. Essa função pode ser usada em um formulário base de toda a sua aplicação para que outros formulários herdem esse método ou pode ser implementado em um método estático global onde se passa um formulário como parâmetro.

Esse método pode ser visto na **Listagem 10**, que mostra também a maneira de se usar.

Listagem 10. Método ObtemValor

```
function TForm1.ObtemValor(const nome: string): string;
var
  controle: IControle;
begin

  if Supports(FindComponent(nome), IControle, Controle) then
    Result := controle.ObtemValor
  else
    Result := 'Null'; //pode ser um string vazio ou um exception

end;

//uso

procedure TForm1.Button7Click(Sender: TObject);
begin
  mControles.Lines.Add(
   ObtemValor( InputBox('Digite o nome de um campo', 'Nome do Campo', ''))
  );
end;
```

Conclusão

Neste artigo foi vista toda a teoria envolvendo interfaces, suas declarações, implementação e operações. Foi visto também um exemplo prático utilizando componentes. No próximo artigo serão vistas mais particularidades a respeito de interfaces e um exemplo com um cenário empresarial. Faremos pedidos e notas fiscais de produtos e serviços utilizando interfaces. Até este momento o leitor será capaz de criar sistemas mais desacoplados e flexíveis com o uso de interfaces para criar componentes intercambiáveis. Até o próximo artigo.

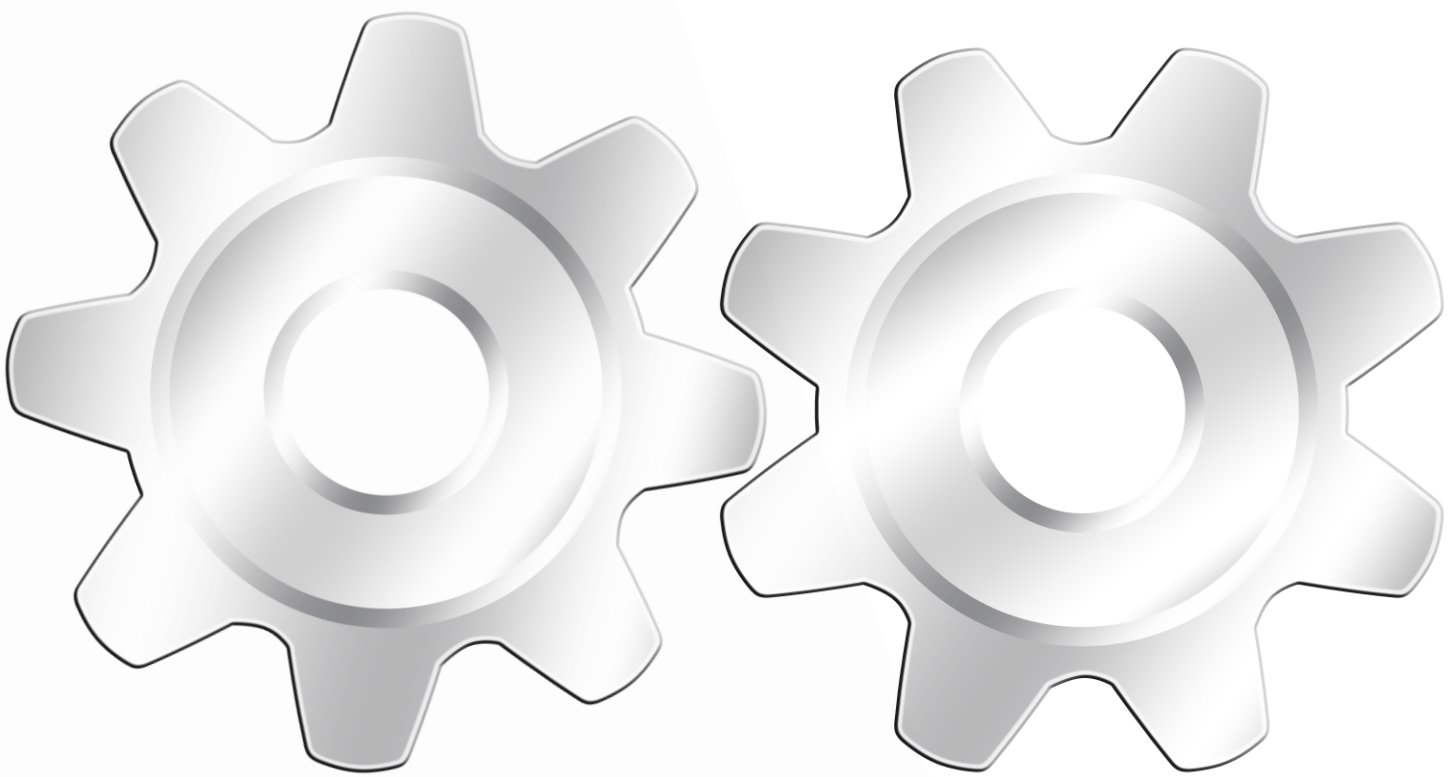
Vitor Luiz Rubio
vitorrubio@gmail.com - http://blog.vitorrubio.com.br/)
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486